

Application Notes

for 80C51-Based 8-Bit Microcontrollers

Philips Semiconductors



FUTURE ELECTRONICS INC.
5935 Airport Road,
Suite 200
Mississauga, Ontario
L4V 1W5
TEL.: (416) 612-9200
FAX: (416) 612-9185
TOLL FREE 1-800-268-7948

PHILIPS

Application Notes for 80C51-Based 8-Bit Microcontrollers

Philips Semiconductors



PHILIPS

Philips Semiconductors and Philips Electronics North America Corporation reserve the right to make changes, without notice, in the products, including circuits, standard cells, and/or software, described or contained herein in order to improve design and/or performance. Philips Semiconductors assumes no responsibility or liability for the use of any of these products, conveys no license or title under any patent, copyright, or mask work right to these products, and makes no representations or warranties that these products are free from patent, copyright, or mask work right infringement, unless otherwise specified. Applications that are described herein for any of these products are for illustrative purposes only. Philips Semiconductors makes no representation or warranty that such applications will be suitable for the specified use without further testing or modification.

LIFE SUPPORT APPLICATIONS

Philips Semiconductors and Philips Electronics North America Corporation Products are not designed for use in life support appliances, devices, or systems where malfunction of a Philips Semiconductors and Philips Electronics North America Corporation Product can reasonably be expected to result in a personal injury. Philips Semiconductors and Philips Electronics North America Corporation customers using or selling Philips Semiconductors and Philips Electronics North America Corporation Products for use in such applications do so at their own risk and agree to fully indemnify Philips Semiconductors and Philips Electronics North America Corporation for any damages resulting from such improper use or sale.

Philips Semiconductors and Philips Electronics North America Corporation register eligible circuits under the Semiconductor Chip Protection Act.

© Copyright Philips Electronics North America Corporation, 1993

Printed in USA

All rights reserved.



Application Notes for 80C51-Based 8-Bit Microcontrollers

Microcontrollers from Philips Semiconductors

Philips Semiconductors supplies a wide range of microcontrollers based on mainstream architectures. By offering a large variety of product derivatives, Philips Semiconductors can meet a broad range of specific or unique application requirements. All of our microcontrollers are based on mainstream architectures to allow the user to take advantage of existing software and a vast array of third-party support.

Philips Semiconductors 8-bit microcontrollers are based on the popular 80C51 architecture. We offer most of the industry standard products in this architecture as well as a large selection of powerful derivative products. These derivatives offer a wide assortment of features, including: additional memory, A/D, PWM, additional timers, and many more. Many of the derivative microcontrollers have an I²C serial interface that allows them to be connected easily to over 70 other parts, increasing their capabilities even further. The I²C serial bus is covered in Section 2 of this book. Philips Semiconductors also offers the Controller Area Network (CAN) serial bus for automotive and industrial applications. This standard, developed by Bosch, offers high noise immunity and error correction for automotive and industrial environments. The CAN serial bus is covered in Section 5 of this book. The Low Power 80CL51 Family of derivatives may be found in Section 4. These devices operate over the wide voltage range of 1.8 - 6.0 volts and are ideal for portable and battery operations. This data book covers the 80C51 standard products and derivatives that Philips Semiconductors manufactures.

Philips Semiconductors 16-bit microcontroller family is based on the 68000 architecture. While these are called 16-bit microcontrollers, the 68000 CPU core architecture is 32-bit. This offers the user a great deal more processing power, when the need arises in a design to move from an 8-bit to a 16-bit microcontroller. Philips Semiconductors 16-bit microcontrollers are software compatible with existing 68000 code. As with our popular 8-bit microcontrollers, EPROM and OTP versions of our 16-bit products are available. The 16-bit microcontrollers are also covered in a separate data book.

Philips Semiconductors is developing a family of 32-bit microcontrollers based on the SPARC RISC architecture. This family of microcontrollers will offer the ultimate in processing power for those applications that are computation intensive in an embedded control environment.

Philips Semiconductors offers uncompromising quality, service, and support with all of its microcontroller products. For a complete family and the best in microcontroller products, look to Philips Semiconductors.

Contents

Application Notes for 80C51-Based 8-Bit Microcontrollers

Preface	iii
80C51 Microcontroller Family Features Guide	vi
Philips Semiconductors Microcontroller Bulletin Boards	viii
CMOS and NMOS 8-Bit Microcontroller Family	ix
CMOS 16-Bit Microcontroller Family	xvi
Section 1 I²C Serial Bus Application Notes & Articles	
AN422 Using the 8XC751 microcontroller as an I ² C bus master	1-3
AN425 Interfacing the PCD8584 I ² C-bus controller to 80C51 family microcontrollers	1-21
AN430 Using the 8XC751/752 in multimaster I ² C applications	1-41
AN433 I ² C slave routines for the 83C751	1-77
AN434 Connecting a PC keyboard to the I ² C-bus	1-83
AN435 Multimaster I ² C routines for byte-oriented I ² C interfaces	1-101
AN438 I ² C routines for 8XC528	1-134
AN444 Using the P82B715 I ² C extender on long cables	1-171
ETV/AN89004 PLM51 I ² C software interface IIC51 (version 0.5)	1-194
EIE/AN91007 I ² C driver routines for 8XC751/2 microcontrollers	1-210
Exploring I ² C	1-276
Programming the I ² C interface	1-280
Section 2 87C750, 8XC751, 8CC752 Application Notes	
AN422 Using the 8XC751 microcontroller as an I ² C bus master	1-3
AN423 Software driven serial communication routines for the 83C751 and 83C752 microcontrollers	2-3
AN426 Controlling air core meters with the 87C751 and SA5775	2-9
AN427 Timer I in non-I ² C applications of the 83/87C751/752 microcontrollers	2-23
AN428 Using the ADC and PWM of the 83C752/87C752	2-29
AN429 Airflow measurement using the 83/87C752 and "C"	2-36
AN430 Using the 8XC751/752 in multimaster I ² C applications	1-41
AN433 I ² C slave routines for the 83C751	1-77
AN436 "Opti-Mizer" power management for notebook computers using the 8XC752 microcontroller	2-56
AN439 87C751 fast NiCad charger	2-66
AN442 (BCM) 87C751 Specification for a bus-controlled monitor	2-78
AN445 ACCESS.bus mouse application code for the microcontroller	2-95
AN446 A software duplex UART for the 751/752	2-125
EIE/AN91007 I ² C driver routines for 8XC751/2 microcontrollers	1-210
Section 3 ACCESS.bus Application Notes & Articles	
AN445 ACCESS.bus mouse application code for the microcontroller	2-95
Serial bus looks to standardize connections for peripherals	3-3
ACCESS.bus, an Open Desktop Bus	3-5
Issues in desktop connectivity	3-12
The Ultimate Desk ACCESSory?	3-16
ACCESS.bus hardware released for industrial and commercial environments	3-22
ACCESS.bus controller board connects 125 peripherals to a single PC/AT comm port	3-24
I/O Standard Gains Multivendor Support	3-25
Special Report: ACCESS.bus Specs And Products	3-26
ACCESS.bus: A New Peripheral Bus	3-28

Contents (Continued)

80C51 microcontroller family features guide

Section 4	Other Application Notes & Articles	
AN408	80C451 operation of port 6	4-3
AN417	256k Centronics printer buffer using the 87C451 microcontroller	4-14
AN418	Counter/timer 2 of the 83C552 microcontroller	4-27
AN420	Using up to 5 external interrupts on 80C51 family microcontrollers	4-34
AN424	8051 family warm boot determinations	4-36
AN440	RAM loader program for 80C51 family applications	4-38
AN443	IEEE Micro Mouse using the 87C751 microcontroller	4-49
AN447	Automatic baud rate detection for the 80C51	4-70
AN448	Determining baud rates for 8051 UARTs and other UART issues	4-73
ESG89001	Electro magnetic compatibility and printed circuit board (PCB) constraints	4-76
EIE/AN91001	Workbench EMC evaluation method	4-104
EIE/AN91006	A/D conversion with P83CL410 PCF1252-x	4-121
EIE/AN91009	Driver for 8xC851 E2PROM	4-139
EIE/AN92001	Low RF-emission applications with a P83CE654 microcontroller	4-162
	Chips push CAN bus into embedded world	4-174
	Add Text Overlay to Any Video Display	4-176

Section 5		North American Sales Offices, Representatives and Distributors	5-3
-----------	--	--	-----

87C550	S	32K	512	3 + Watchdog	4	UART, ISO-R	2	Large Memory for high level languages
88C55A (80C55A)	H	25K	512	3 + Watchdog	4	UART, ISO-R	2	Large Memory for high level languages
87C552	H	16K	512	3 + Watchdog	4	UART, CAN	8	(see above)
87C553	H	16K	512	3 + Watchdog	4	UART, ISO-R	2	
87C554	S	16K	512	3 + Watchdog	4	UART, ISO-R	2	
87C555	S	16K	512	3 + Watchdog	4	UART, ISO-R	2	
87C556	S	16K	512	3 + Watchdog	4	UART, ISO-R	2	
87C557	S	16K	512	3 + Watchdog	4	UART, ISO-R	2	
87C558	S	16K	512	3 + Watchdog	4	UART, ISO-R	2	
87C559	S	16K	512	3 + Watchdog	4	UART, ISO-R	2	
87C560	S	16K	512	3 + Watchdog	4	UART, ISO-R	2	
87C561	S	16K	512	3 + Watchdog	4	UART, ISO-R	2	
87C562	S	16K	512	3 + Watchdog	4	UART, ISO-R	2	
87C563	S	16K	512	3 + Watchdog	4	UART, ISO-R	2	
87C564	S	16K	512	3 + Watchdog	4	UART, ISO-R	2	
87C565	S	16K	512	3 + Watchdog	4	UART, ISO-R	2	
87C566	S	16K	512	3 + Watchdog	4	UART, ISO-R	2	
87C567	S	16K	512	3 + Watchdog	4	UART, ISO-R	2	
87C568	S	16K	512	3 + Watchdog	4	UART, ISO-R	2	
87C569	S	16K	512	3 + Watchdog	4	UART, ISO-R	2	
87C570	S	16K	512	3 + Watchdog	4	UART, ISO-R	2	
87C571	S	16K	512	3 + Watchdog	4	UART, ISO-R	2	
87C572	S	16K	512	3 + Watchdog	4	UART, ISO-R	2	
87C573	S	16K	512	3 + Watchdog	4	UART, ISO-R	2	
87C574	S	16K	512	3 + Watchdog	4	UART, ISO-R	2	
87C575	S	16K	512	3 + Watchdog	4	UART, ISO-R	2	
87C576	S	16K	512	3 + Watchdog	4	UART, ISO-R	2	
87C577	S	16K	512	3 + Watchdog	4	UART, ISO-R	2	
87C578	S	16K	512	3 + Watchdog	4	UART, ISO-R	2	
87C579	S	16K	512	3 + Watchdog	4	UART, ISO-R	2	
87C580	S	16K	512	3 + Watchdog	4	UART, ISO-R	2	
87C581	S	16K	512	3 + Watchdog	4	UART, ISO-R	2	
87C582	S	16K	512	3 + Watchdog	4	UART, ISO-R	2	
87C583	S	16K	512	3 + Watchdog	4	UART, ISO-R	2	
87C584	S	16K	512	3 + Watchdog	4	UART, ISO-R	2	
87C585	S	16K	512	3 + Watchdog	4	UART, ISO-R	2	
87C586	S	16K	512	3 + Watchdog	4	UART, ISO-R	2	
87C587	S	16K	512	3 + Watchdog	4	UART, ISO-R	2	
87C588	S	16K	512	3 + Watchdog	4	UART, ISO-R	2	
87C589	S	16K	512	3 + Watchdog	4	UART, ISO-R	2	
87C590	S	16K	512	3 + Watchdog	4	UART, ISO-R	2	
87C591	S	16K	512	3 + Watchdog	4	UART, ISO-R	2	
87C592	S	16K	512	3 + Watchdog	4	UART, ISO-R	2	
87C593	S	16K	512	3 + Watchdog	4	UART, ISO-R	2	
87C594	S	16K	512	3 + Watchdog	4	UART, ISO-R	2	
87C595	S	16K	512	3 + Watchdog	4	UART, ISO-R	2	
87C596	S	16K	512	3 + Watchdog	4	UART, ISO-R	2	
87C597	S	16K	512	3 + Watchdog	4	UART, ISO-R	2	
87C598	S	16K	512	3 + Watchdog	4	UART, ISO-R	2	
87C599	S	16K	512	3 + Watchdog	4	UART, ISO-R	2	
87C600	S	16K	512	3 + Watchdog	4	UART, ISO-R	2	
87C601	S	16K	512	3 + Watchdog	4	UART, ISO-R	2	
87C602	S	16K	512	3 + Watchdog	4	UART, ISO-R	2	
87C603	S	16K	512	3 + Watchdog	4	UART, ISO-R	2	
87C604	S	16K	512	3 + Watchdog	4	UART, ISO-R	2	
87C605	S	16K	512	3 + Watchdog	4	UART, ISO-R	2	
87C606	S	16K	512	3 + Watchdog	4	UART, ISO-R	2	
87C607	S	16K	512	3 + Watchdog	4	UART, ISO-R	2	
87C608	S	16K	512	3 + Watchdog	4	UART, ISO-R	2	
87C609	S	16K	512	3 + Watchdog	4	UART, ISO-R	2	
87C610	S	16K	512	3 + Watchdog	4	UART, ISO-R	2	
87C611	S	16K	512	3 + Watchdog	4	UART, ISO-R	2	
87C612	S	16K	512	3 + Watchdog	4	UART, ISO-R	2	
87C613	S	16K	512	3 + Watchdog	4	UART, ISO-R	2	
87C614	S	16K	512	3 + Watchdog	4	UART, ISO-R	2	
87C615	S	16K	512	3 + Watchdog	4	UART, ISO-R	2	
87C616	S	16K	512	3 + Watchdog	4	UART, ISO-R	2	
87C617	S	16K	512	3 + Watchdog	4	UART, ISO-R	2	
87C618	S	16K	512	3 + Watchdog	4	UART, ISO-R	2	
87C619	S	16K	512	3 + Watchdog	4	UART, ISO-R	2	
87C620	S	16K	512	3 + Watchdog	4	UART, ISO-R	2	
87C621	S	16K	512	3 + Watchdog	4	UART, ISO-R	2	
87C622	S	16K	512	3 + Watchdog	4	UART, ISO-R	2	
87C623	S	16K	512	3 + Watchdog	4	UART, ISO-R	2	
87C624	S	16K	512	3 + Watchdog	4	UART, ISO-R	2	
87C625	S	16K	512	3 + Watchdog	4	UART, ISO-R	2	
87C626	S	16K	512	3 + Watchdog	4	UART, ISO-R	2	
87C627	S	16K	512	3 + Watchdog	4	UART, ISO-R	2	
87C628	S	16K	512	3 + Watchdog	4	UART, ISO-R	2	
87C629	S	16K	512	3 + Watchdog	4	UART, ISO-R	2	
87C630	S	16K	512	3 + Watchdog	4	UART, ISO-R	2	
87C631	S	16K	512	3 + Watchdog	4	UART, ISO-R	2	
87C632	S	16K	512	3 + Watchdog	4	UART, ISO-R	2	
87C633	S	16K	512	3 + Watchdog	4	UART, ISO-R	2	
87C634	S	16K	512	3 + Watchdog	4	UART, ISO-R	2	
87C635	S	16K	512	3 + Watchdog	4	UART, ISO-R	2	
87C636	S	16K	512	3 + Watchdog	4	UART, ISO-R	2	
87C637	S	16K	512	3 + Watchdog	4	UART, ISO-R	2	
87C638	S	16K	512	3 + Watchdog	4	UART, ISO-R	2	
87C639	S	16K	512	3 + Watchdog	4	UART, ISO-R	2	
87C640	S	16K	512	3 + Watchdog	4	UART, ISO-R	2	
87C641	S	16K	512	3 + Watchdog	4	UART, ISO-R	2	
87C642	S	16K	512	3 + Watchdog	4	UART, ISO-R	2	
87C643	S	16K	512	3 + Watchdog	4	UART, ISO-R	2	
87C644	S	16K	512	3 + Watchdog	4	UART, ISO-R	2	
87C645	S	16K	512	3 + Watchdog	4	UART, ISO-R	2	
87C646	S	16K	512	3 + Watchdog	4	UART, ISO-R	2	
87C647	S	16K	512	3 + Watchdog	4	UART, ISO-R	2	
87C648	S	16K	512	3 + Watchdog	4	UART, ISO-R	2	
87C649	S	16K	512	3 + Watchdog	4	UART, ISO-R	2	
87C650	S	16K	512	3 + Watchdog	4	UART, ISO-R	2	
87C651	S	16K	512	3 + Watchdog	4	UART, ISO-R	2	
87C652	S	16K	512	3 + Watchdog	4	UART, ISO-R	2	
87C653	S	16K	512	3 + Watchdog	4	UART, ISO-R	2	
87C654	S	16K	512	3 + Watchdog	4	UART, ISO-R	2	
87C655	S	16K	512	3 + Watchdog	4	UART, ISO-R	2	
87C656	S	16K	512	3 + Watchdog	4	UART, ISO-R	2	
87C657	S	16K	512	3 + Watchdog	4	UART, ISO-R	2	
87C658	S	16K	512	3 + Watchdog	4	UART, ISO-R	2	
87C659	S	16K	512	3 + Watchdog	4	UART, ISO-R	2	
87C660	S	16K	512	3 + Watchdog	4	UART, ISO-R	2	
87C661	S	16K	512	3 + Watchdog	4	UART, ISO-R	2	
87C662	S	16K	512	3 + Watchdog	4	UART, ISO-R	2	
87C663	S	16K	512	3 + Watchdog	4	UART, ISO-R	2	
87C664	S	16K	512	3 + Watchdog	4	UART, ISO-R	2	
87C665	S	16K	512	3 + Watchdog	4	UART, ISO-R	2	
87C666	S	16K	512	3 + Watchdog	4	UART, ISO-R	2	
87C667	S	16K	512	3 + Watchdog	4	UART, ISO-R	2	
87C668	S	16K	512	3 + Watchdog	4	UART, ISO-R	2	
87C669	S	16K	512	3 + Watchdog	4	UART, ISO-R	2	
87C670	S	16K	512	3 + Watchdog	4	UART, ISO-R	2	
87C671	S	16K	512	3 + Watchdog	4	UART, ISO-R	2	
87C672	S	16K	512	3 + Watchdog	4	UART, ISO-R	2	
87C673	S	16K	512	3 + Watchdog	4	UART, ISO-R	2	
87C674	S	16K	512	3 + Watchdog	4	UART, ISO-R	2	
87C675	S	16K	512	3 + Watchdog	4	UART, ISO-R	2	
87C676	S	16K	512	3 + Watchdog	4	UART, ISO-R	2	
87C677	S	16K	512	3 + Watchdog	4	UART, ISO-R	2	
87C678	S	16K	512	3 + Watchdog	4	UART, ISO-R	2	
87C679	S	16K	512	3 + Watchdog	4	UART, ISO-R	2	
87C680	S	16K	512	3 + Watchdog	4	UART, ISO-R	2	
87C681	S	16K	512	3 + Watchdog	4	UART, ISO-R	2	
87C682	S	16K	512	3 + Watchdog	4	UART, ISO-R	2	
87C683	S	16K	512	3 + Watchdog	4	UART, ISO-R	2	
87C684	S	16K	512	3 + Watchdog	4	UART, ISO-R	2	
87C685	S	16K	512	3 + Watchdog	4	UART, ISO-R	2	
87C686	S	16K	512	3 + Watchdog	4	UART, ISO-R	2	
87C687	S	16K	512	3 + Watchdog	4	UART, ISO-R	2	
87C688	S	16K	512	3 + Watchdog	4	UART, ISO-R	2	
87C689	S	16K	512	3 + Watchdog	4	UART, ISO-R	2	
87C690	S	16K	512	3 + Watchdog	4	UART, ISO-R	2	
87C691	S	16K	512	3 + Watchdog	4	UART, ISO-R	2	
87C692	S	16K	512	3 + Watchdog	4	UART, ISO-R	2	
87C693	S	16K	512	3 + Watchdog	4	UART, ISO-R	2	
87C694	S	16K	512	3 + Watchdog	4	UART, ISO-R	2	
87C695	S	16K	512	3 + Watchdog	4	UART, ISO-R	2	
87C696	S	16K	512	3 + Watchdog	4	UART, ISO-R	2	
87C697	S	16K	512	3 + Watchdog	4	UART, ISO-R	2	
87C698	S	16K	512	3 + Watchdog	4	UART, ISO-R	2	
87C699	S	16K	512	3 + Watchdog	4	UART, ISO-R	2	
87C700	S	16K	512	3 + Watchdog	4	UART, ISO-R	2	
87C701	S	16K	512	3 + Watchdog	4	UART, ISO-R	2	
87C702	S	16K	512	3 + Watchdog	4	UART, ISO-R	2	
87C703	S	16K	512	3 + Watchdog	4	UART, ISO-R	2	
87C704	S	16K	512	3 + Watchdog	4	UART, ISO-R	2	
87C705	S	16K	512	3 + Watchdog	4	UART, ISO-R	2	
87C706	S	16K	512	3 + Watchdog	4	UART, ISO-R	2	
87C707	S	16K	512	3 + Watchdog	4	UART, ISO-R	2	
87C708	S	16K	512	3 + Watchdog	4	UART, ISO-R	2	
87C709	S	16K	512	3 + Watchdog	4	UART, ISO-R	2	
87C710	S	16K	512	3 + Watchdog	4	UART, ISO-R	2	
87C711	S	16K	512	3 + Watchdog	4	UART, ISO-R	2	
87C712	S	16K	512	3 + Watchdog	4	UART, ISO-R	2	
87C713	S	16K	512	3 + Watchdog	4	UART, ISO-R	2	
87C714	S	16K	512	3 + Watchdog	4	UART, ISO-R	2	
87C715	S	16K	512	3 + Watchdog	4	UART, ISO-R	2	
87C716	S	16K	512	3 + Watchdog	4	UART, ISO-R	2	
87C717	S	16K	512	3 + Watchdog	4	UART, ISO-R	2	
87C718	S	16K	512	3 + Watchdog	4	UART, ISO-R	2	
87C719	S	16K	512	3 + Watchdog	4	UART, ISO-R	2	
87C720	S	16K	512	3 + Watchdog	4	UART, ISO-R	2	
87C721	S	16K	512	3 + Watchdog	4	UART, ISO-R	2	
87C722	S	16K	512	3 + Watchdog	4	UART, ISO-R	2	
87C723	S	16K	512	3 + Watchdog	4	UART, ISO-R	2	
87C724	S	16K	512</					

80C51 microcontroller family features guide

Microcontroller Parts Characteristic Guide; (part 1)

Part Number (ROMless)		Memory			Counter Timers	I/O Ports	Serial Interfaces	External Interrupt	Comments/ Special Features
		ROM	EPROM	RAM					
87C750	S		1K	64	1 (16-bit)	2-3/8	-	2	Lowest cost, 24-pin Skinny DIP
83C751	S	2K		64	1 (16-bit)	2-3/8	I2C (bit)	2	Low-Cost 24-pin Skinny DIP
87C751	S		2K	64	1 (16-bit)	2-3/8	I2C (bit)	2	Low-Cost 24-pin Skinny DIP
83C752	S	2K		64	1 (16-bit)	2-5/8	I2C (bit)	2	5 Channel 8-bit A/D, PWM Output
87C752	S		2K	64	1 (16-bit)	2-5/8	I2C (bit)	2	5 Channel 8-bit A/D, PWM Output
8051AH (8031AH)	S	4K		128	2	4	UART	2	NMOS
SC80C51 (80C31)	S	4K		128	2	4	UART	2	CMOS (Sunnyvale)
PCx80C51 (80C31)	H	4K		128	2	4	UART	2	CMOS (Hamburg)
87C51	S		4K	128	2	4	UART	2	CMOS
80CL51 (80CL31)	Z	4K		128	2	4	UART	10	Low Voltage (1.8V to 6V), Low Power
83CL410 (80CL410)	Z	4K		128	2	4	I2C	10	Low Voltage (1.8V to 6V), Low Power
83C451 (80C451)	S	4K		128	2	7	UART	2	Extended I/O, Processor Bus Interface
87C451	S		4K	128	2	7	UART	2	Extended I/O, Processor Bus Interface
83C550 (80C550)	S	4K		128	2 + Watchdog	4	UART	2	8 Channel 8-bit A/D
87C550	S		4K	128	2 + Watchdog	4	UART	2	8 Channel 8-bit A/D
83C851 (80C851)	H	4K		128	2	4	UART	2	256B EEPROM, 80C51 Pin compatible
83C852	H	6K		256	2 (16-bit)	2/8	-	1	Smartcard Controller with 2K EEPROM (Data, Code) Cryptographic Calc Unit
83CL580	Z	6K		256	3 + Watchdog	5	UART, I2C	10	4 Channel 8-bit A/D, PWM Output, Low Voltage (2.5V to 6V), Low Power
8052AH (8032AH)	S	8K		256	3	4	UART	2	NMOS
80C52 (80C32)	S	8K		256	3	4	UART	2	80C51 Pin Compatible
87C52	S		8K	256	3	4	UART	2	(see above)
80CL52 (80CL32)	Z	8K		256	3	4	UART	2	Low Voltage (1.8V to 6V), Low Power
83C652 (80C652)	H	8K		256	2	4	UART, I2C	2	80C51 Pin Compatible
87C652	S		8K	256	2	4	UART, I2C	2	(see above)
83C575 (80C575)	S	8K		256	3 + PCA + Watchdog	4	UART	2	High Reliability, with Low Voltage Detect, Osc Fail Detect, Analog Comparators, PCA
87C575	S		8K	256	(see above)	4	UART	2	(see above)
83C552 (80C552)	H	8K		256	3 + Watchdog	6	UART, I2C	2	8 Channel 10-bit A/D, 2 PWM Outputs, Capture/Compare Timer
87C552	S		8K	256	3 + Watchdog	6	UART, I2C	2	(see above)
83C562 (80C562)	H	8K		256	3 + Watchdog	6	UART	2	8 Channel 8-bit A/D, 2 PWM Outputs, Capture/Compare Timer
83C053	S	8K		192	2 (16-bit)	3.5	-	2	On-Screen Display, 9 PWM Outputs, 3 Software A/D Inputs
83C054	S	16K		192	2 (16-bit)	3.5	-	2	(see above)
87C054	S		16K	192	2 (16-bit)	3.5	-	2	(see above)
87C055	S		16K	256	2 (16-bit)	3.5	-	2	(see above, extra RAM added)
83C654	H	16K		256	2	4	UART, I2C	2	80C51 Pin Compatible
87C654	S		16K	256	2	4	UART, I2C	2	(see above)
83CE654	H	16K		256	2	4	UART, I2C	2	83C654 with Reduced EMI
83CL781	Z	16K		256	3	4	UART, I2C	10	Low Voltage (1.8V to 6V), Low Power
83CL782	Z	16K		256	3	4	UART, I2C	10	83CL781 Optimized 12MHz @ 3.1V
87C51FB	S		16K	256	3 + PCA	4	UART	2	Enhanced UART, 3 timers + PCA
83C524	H	16K		512	3 + Watchdog	4	UART, I2C-bit	2	512 RAM
87C524	S		16K	512	3 + Watchdog	4	UART, I2C-bit	2	512 RAM
83C592 (80C592)	H	16K		512	3 + Watchdog	6	UART, CAN	6	CAN Bus Controller with 8 x 10-bit A/D, 2 PWM outputs, Capture/Compare Timer
87C592	H		16K	512	3 + Watchdog	6	UART, CAN	6	(see above)
83C528 (80C528)	H	32K		512	3 + Watchdog	4	UART, I2C-bit	2	Large Memory for High Level Languages
87C528	S		32K	512	3 + Watchdog	4	UART, I2C-bit	2	Large Memory for High Level Languages

Note: Production Centers are indicated in the second column: H - Hamburg, S - Sunnyvale, Z - Zurich
All combinations of part type, speed, temperature and package may not be available.

80C51 microcontroller family features guide

Microcontroller Parts Characteristic Guide; (part 2)

Part Number (ROMless)	Program Security?	Clock Frequency (MHz)	Temperature Ranges (°C)			Package				
			0 to 70	-40 to +85	-55 to +125	PDIP	CDIP	PLCC	CLCC	PQFP
87C750	S	Y	3.5 to 40	X	X	N24	F24	A28		
83C751	S	N	3.5 to 16	X	X	N24		A28		
87C751	S	Y	3.5 to 16	X	X	N24	F24	A28		
83C752	S	N	3.5 to 16	X	X	N28		A28		
87C752	S	Y	3.5 to 16	X	X	N28	F28	A28		
8051AH (8031AH)	S	N	3.5 to 15	X	X	N40		A44		
SC80C51 (80C31)	S	Y	0.5 to 33	X	X	N40		A44		B44
PCx80C51 (80C31)	H	N	1.2 to 30	X	X	P (40)		WP (44)		H (44)
87C51	S	Y	0.5 to 33	X	X	N40	F40	A44	K44	B44
80CL51 (80CL31)	Z	N	0 to 16 (1)		X	N40 (2)				B44
83CL410 (80CL410)	Z	N	0 to 16 (1)		X	N40 (2)				
83C451 (80C451)	S	N	3.5 to 16	X	X	N64		A68	L68	
87C451	S	Y	3.5 to 16	X	X	N64		A68	L68	
83C550 (80C550)	S	Y	3.5 to 16	X	X	N40		A44		
87C550	S	Y	3.5 to 16	X	X	N40	F40	A44	K44	
83C851 (80C851)	H	Y	1.2 to 16	X	X	N40		A44		B44
83C852	H	Y	1 to 12	X		die only				
83CL580	Z	N	0 to 12 (1)		X	(3)				B64
8052AH (8032AH)	S	N	3.5 to 15	X	X	N40		A44		
80C52 (80C32)	S	Y	3.5 to 24	X	X	N40		A44		B44
87C52	S	Y	3.5 to 24	X	X	N40	F40	A44	K44	B44
80CL52 (80CL32)	Z	N	0 to 12 (1)		X	N40				B44
83C652 (80C652)	H	Y	1.2 to 24	X	X	N40		A44		B44
87C652	S	Y	1.2 to 20	X	X	N40	F40	A44	K44	
83C575 (80C575)	S	Y	4 to 16	X		N40		A44		B44
87C575	S	Y	4 to 16	X		N40	F40	A44	K44	B44
83C552 (80C552)	H	N	1.2 to 30	X	X			A68		B80
87C552	S	Y	1.2 to 16	X				A68	K68	
83C562 (80C562)	H	N	1.2 to 16	X	X			A68		
83C053	S	N	3.5 to 12	X		42 SDIP				
83C054	S	N	3.5 to 12	X		42 SDIP				
87C054	S	N	3.5 to 12	X		42 SDIP				
87C055	S	N	3.5 to 20	X		42 SDIP				
83C654	H	Y	1.2 to 24	X	X	N40		A44		B44
87C654	S	Y	1.2 to 20	X	X	N40	F40	A44	K44	B44
83CE654	H	Y	1.2 to 16	X	X					B44
83CL781	Z	N	0 to 12 (1)		X	N40				B44
83CL782	Z	N	0 to 12 (1)		-25 to +55	N40				B44
87C51FB	S	Y	3.5 to 16	X	X	N40	F40	A44	K44	B44
83C524	H	Y	1.2 to 16	X	X	N40		A44		B44
87C524	S	Y	3.5 to 12	X	X	N40	F40	A44	K44	B44
83C592 (80C592)	H	Y	1.2 to 16		X			A68	K68	
87C592	H	Y	1.2 to 16	X				A68	K68	
83C528 (80C528)	H	Y	1.2 to 16	X	X	N40		A44		B44
87C528	S	Y	3.5 to 20	X	X	N40	F40	A44	K44	B44

Note: 1) Oscillator options start from 32kHz.
VSO56 Package.

2) Also available in VSO40 package.

3) Also available in

Microcontroller bulletin boards

To better serve our customers, Philips maintains a microcontroller bulletin board. This computer bulletin board system features a microcontroller newsletter, application and demonstration programs for download, and the ability to send messages to microcontroller application engineers. The system can be accessed with a modem at 2400, 1200, or 300 baud.

The telephone numbers are:

**(800) 451-6644 (in the U.S.)
or
(408) 991-2406**

Please call us anytime!

We also have a ROM code bulletin board through which you can submit ROM codes. This is a closed bulletin board for security reasons. To get an ID, contact your local sales office. The system can be accessed with a 2400, 1200, or 300 baud modem, and is available 24 hours a day.

The telephone number is:

(408) 991-3459

Part Number (ROM)	Program Security?	Clock Frequency (MHz)	0 to 10	10 to 40	40 to 100	100 to 250	250 to 500	500 to 1000	1000 to 2000	2000 to 4000	4000 to 8000	8000 to 16000	16000 to 32000	32000 to 64000	64000 to 128000	128000 to 256000	256000 to 512000	512000 to 1024000	1024000 to 2048000	2048000 to 4096000	4096000 to 8192000	8192000 to 16384000	16384000 to 32768000	32768000 to 65536000	65536000 to 131072000	131072000 to 262144000	262144000 to 524288000	524288000 to 1048576000	1048576000 to 2097152000	2097152000 to 4194304000	4194304000 to 8388608000	8388608000 to 16777216000	16777216000 to 33554432000	33554432000 to 67108864000	67108864000 to 134217728000	134217728000 to 268435456000	268435456000 to 536870912000	536870912000 to 1073741824000	1073741824000 to 2147483648000	2147483648000 to 4294967296000	4294967296000 to 8589934592000	8589934592000 to 17179869184000	17179869184000 to 34359738368000	34359738368000 to 68719476736000	68719476736000 to 137438953472000	137438953472000 to 274877906944000	274877906944000 to 549755813888000	549755813888000 to 1099511627776000	1099511627776000 to 2199023255552000	2199023255552000 to 4398046511104000	4398046511104000 to 8796093022208000	8796093022208000 to 17592186044416000	17592186044416000 to 35184372088832000	35184372088832000 to 70368744177664000	70368744177664000 to 140737488355328000	140737488355328000 to 281474976710656000	281474976710656000 to 562949953421312000	562949953421312000 to 1125899906842624000	1125899906842624000 to 2251799813685248000	2251799813685248000 to 4503599627370496000	4503599627370496000 to 9007199254740992000	9007199254740992000 to 18014398509481984000	18014398509481984000 to 36028797018963968000	36028797018963968000 to 72057594037927936000	72057594037927936000 to 144115188075855872000	144115188075855872000 to 288230376151711744000	288230376151711744000 to 576460752303423488000	576460752303423488000 to 1152921504606846976000	1152921504606846976000 to 2305843009213693952000	2305843009213693952000 to 4611686018427387904000	4611686018427387904000 to 9223372036854775808000	9223372036854775808000 to 18446744073709551616000	18446744073709551616000 to 36893488147419103232000	36893488147419103232000 to 73786976294838206464000	73786976294838206464000 to 147573952589676412928000	147573952589676412928000 to 295147905179352825856000	295147905179352825856000 to 590295810358705651712000	590295810358705651712000 to 1180591620717411303424000	1180591620717411303424000 to 2361183241434822606848000	2361183241434822606848000 to 4722366482869645213696000	4722366482869645213696000 to 9444732965739290427392000	9444732965739290427392000 to 18889465931478580854784000	18889465931478580854784000 to 37778931862957161709568000	37778931862957161709568000 to 75557863725914323419136000	75557863725914323419136000 to 151115727451828646838272000	151115727451828646838272000 to 302231454903657293676544000	302231454903657293676544000 to 604462909807314587353088000	604462909807314587353088000 to 1208925819614629174706176000	1208925819614629174706176000 to 2417851639229258349412352000	2417851639229258349412352000 to 4835703278458516698824704000	4835703278458516698824704000 to 9671406556917033397649408000	9671406556917033397649408000 to 19342813113834066795298816000	19342813113834066795298816000 to 38685626227668133590597632000	38685626227668133590597632000 to 77371252455336267181195264000	77371252455336267181195264000 to 154742504910672534362390528000	154742504910672534362390528000 to 309485009821345068724781056000	309485009821345068724781056000 to 618970019642690137449562112000	618970019642690137449562112000 to 1237940039285380274899124224000	1237940039285380274899124224000 to 2475880078570760549798248448000	2475880078570760549798248448000 to 4951760157141521099596496896000	4951760157141521099596496896000 to 9903520314283042199192993792000	9903520314283042199192993792000 to 19807040628566084398385987584000	19807040628566084398385987584000 to 39614081257132168796771975168000	39614081257132168796771975168000 to 79228162514264337593543950336000	79228162514264337593543950336000 to 158456325028528675187087900672000	158456325028528675187087900672000 to 316912650057057350374175801344000	316912650057057350374175801344000 to 633825300114114700748351602688000	633825300114114700748351602688000 to 1267650600228229401496703205376000	1267650600228229401496703205376000 to 2535301200456458802993406410752000	2535301200456458802993406410752000 to 5070602400912917605986812821504000	5070602400912917605986812821504000 to 10141204801825835211973625643008000	10141204801825835211973625643008000 to 20282409603651670423947251286016000	20282409603651670423947251286016000 to 40564819207303340847894502572032000	40564819207303340847894502572032000 to 81129638414606681695789005144064000	81129638414606681695789005144064000 to 162259276829213363391578010288128000	162259276829213363391578010288128000 to 324518553658426726783156020576256000	324518553658426726783156020576256000 to 649037107316853453566312041152512000	649037107316853453566312041152512000 to 1298074214633706907132624082305024000	1298074214633706907132624082305024000 to 2596148429267413814265248164610048000	2596148429267413814265248164610048000 to 5192296858534827628530496329220096000	5192296858534827628530496329220096000 to 10384593717069655257060992658440192000	10384593717069655257060992658440192000 to 20769187434139310514121985316880384000	20769187434139310514121985316880384000 to 41538374868278621028243970633760768000	41538374868278621028243970633760768000 to 83076749736557242056487941267521536000	83076749736557242056487941267521536000 to 166153499473114484112975882535043072000	166153499473114484112975882535043072000 to 332306998946228968225951765070086144000	332306998946228968225951765070086144000 to 664613997892457936451903530140172288000	664613997892457936451903530140172288000 to 1329227995784915872903807060280344576000	1329227995784915872903807060280344576000 to 2658455991569831745807614120560689152000	2658455991569831745807614120560689152000 to 5316911983139663491615228241121378304000	5316911983139663491615228241121378304000 to 10633823966279326983230456482242756608000	10633823966279326983230456482242756608000 to 21267647932558653966460912964485513216000	21267647932558653966460912964485513216000 to 42535295865117307932921825928971026432000	42535295865117307932921825928971026432000 to 85070591730234615865843651857942052864000	85070591730234615865843651857942052864000 to 170141183460469231731687303715884105728000	170141183460469231731687303715884105728000 to 340282366920938463463374607431768211456000	340282366920938463463374607431768211456000 to 680564733841876926926749214863536422912000	680564733841876926926749214863536422912000 to 1361129467683753853853498429727072845824000	1361129467683753853853498429727072845824000 to 2722258935367507707706996859454145691648000	2722258935367507707706996859454145691648000 to 5444517870735015415413993718908291383296000	5444517870735015415413993718908291383296000 to 10889035741470030830827987437816582766592000	10889035741470030830827987437816582766592000 to 21778071482940061661655974875633165533184000	21778071482940061661655974875633165533184000 to 43556142965880123323311949751266331066368000	43556142965880123323311949751266331066368000 to 87112285931760246646623899502532662132736000	87112285931760246646623899502532662132736000 to 174224571863520493293247799005065324265472000	174224571863520493293247799005065324265472000 to 348449143727040986586495598010130648530944000	348449143727040986586495598010130648530944000 to 696898287454081973172991196020261297061888000	696898287454081973172991196020261297061888000 to 1393796574908163946345982392040522594123776000	1393796574908163946345982392040522594123776000 to 2787593149816327892691964784081045188247552000	2787593149816327892691964784081045188247552000 to 5575186299632655785383929568162090376495104000	5575186299632655785383929568162090376495104000 to 11150372599265311570767859136324180752990208000	11150372599265311570767859136324180752990208000 to 22300745198530623141535718272648361505980416000	22300745198530623141535718272648361505980416000 to 44601490397061246283071436545296723011960832000	44601490397061246283071436545296723011960832000 to 89202980794122492566142873090593446023921664000	89202980794122492566142873090593446023921664000 to 178405961588244985132285746181186892047843328000	178405961588244985132285746181186892047843328000 to 356811923176489970264571492362373784095686656000	356811923176489970264571492362373784095686656000 to 713623846352979940529142984724747568191373312000	713623846352979940529142984724747568191373312000 to 1427247692705959881058285969449495136382746624000	1427247692705959881058285969449495136382746624000 to 2854495385411919762116571938898990272765493248000	2854495385411919762116571938898990272765493248000 to 5708990770823839524233143877797980545530986496000	5708990770823839524233143877797980545530986496000 to 11417981541647679048466287755595961091061972992000	11417981541647679048466287755595961091061972992000 to 22835963083295358096932575511191922182123945984000	22835963083295358096932575511191922182123945984000 to 45671926166590716193865151022383844364247891968000	45671926166590716193865151022383844364247891968000 to 91343852333181432387730302044767688728495783936000	91343852333181432387730302044767688728495783936000 to 182687704666362864775460604089535377456991567872000	182687704666362864775460604089535377456991567872000 to 365375409332725729550921208179070754913983135744000	365375409332725729550921208179070754913983135744000 to 730750818665451459101842416358141509827966271488000	730750818665451459101842416358141509827966271488000 to 1461501637330902918203684832716283019655932542976000	1461501637330902918203684832716283019655932542976000 to 2923003274661805836407369665432566039311865085952000	2923003274661805836407369665432566039311865085952000 to 5846006549323611672814739330865132078623730171904000	5846006549323611672814739330865132078623730171904000 to 11692013098647223345629478661730264157247460343808000	11692013098647223345629478661730264157247460343808000 to 23384026197294446691258957323460528314494920687616000	23384026197294446691258957323460528314494920687616000 to 46768052394588893382517914646921048988989841375232000	46768052394588893382517914646921048988989841375232000 to 93536104789177786765035829293842097977979682750464000	93536104789177786765035829293842097977979682750464000 to 187072209578355573530071658587684195955959365500928000	187072209578355573530071658587684195955959365500928000 to 374144419156711147060143317175368391911918731001856000	374144419156711147060143317175368391911918731001856000 to 748288838313422294120286634350736783823837462003712000	748288838313422294120286634350736783823837462003712000 to 1496577676626844588240573268701473567647674924007424000	1496577676626844588240573268701473567647674924007424000 to 2993155353253689176481146537402947135295349848014848000	2993155353253689176481146537402947135295349848014848000 to 5986310706507378352962293074805894270590699696029696000	5986310706507378352962293074805894270590699696029696000 to 11972621413014756705924586149611788541181399392059392000	11972621413014756705924586149611788541181399392059
-------------------	-------------------	-----------------------	---------	----------	-----------	------------	------------	-------------	--------------	--------------	--------------	---------------	----------------	----------------	-----------------	------------------	------------------	-------------------	--------------------	--------------------	--------------------	---------------------	----------------------	----------------------	-----------------------	------------------------	------------------------	-------------------------	--------------------------	--------------------------	--------------------------	---------------------------	----------------------------	----------------------------	-----------------------------	------------------------------	------------------------------	-------------------------------	--------------------------------	--------------------------------	--------------------------------	---------------------------------	----------------------------------	----------------------------------	-----------------------------------	------------------------------------	------------------------------------	-------------------------------------	--------------------------------------	--------------------------------------	--------------------------------------	---------------------------------------	--	--	---	--	--	---	--	--	--	---	--	--	---	--	--	---	--	--	--	---	--	--	---	--	--	---	--	--	--	---	--	--	---	--	--	---	--	--	--	---	--	--	---	--	--	---	--	--	--	---	--	--	---	--	--	---	--	--	---	--	--	--	---	--	--	---	--	--	---	--	--	--	---	--	--	---	--	--	---	--	--	--	---	--	--	---	--	--	---	--	--	--	---	--	--	---	--	--	---	--	--	--	---	--	--	---	--	--	---	--	--	--	---	--	--	---	--	--	---	--	--	--	---	--	--	---	--	--	---	--

CMOS and NMOS 8-bit microcontroller family

80C51 FAMILY CMOS

(Continued)

TYPE	ROM/ EPROM	RAM	SPEED (MHz)	PACKAGE	FUNCTIONS	REMARKS	PROBE SDS	THIRD PARTY EMULATOR	REMARKS
80C31 80C51 87C51	0 4k ROM 4k EPROM	128 128 128	33 33 33	DIL40, LCC44 QFP44	UART, 2 timers		OM1092 + OM1097 (16MHz)	8052PC(M) POD-C51B(N)	OM1092: Universal probe OM1095: Upgrade unit
80C32 80C52 87C52	0 8k ROM 8k EPROM	256 256 256	20 20 20	DIL40, LCC44 QFP44	UART, 3 timers		OM4111 + OM4110	8052PC(M) POD-C32(N)	
80C451 83C451 87C451	0 4k ROM 4k EPROM	128 128 128	16 16 16	DIP64/LCC68	UART, 2 timers Extended I/O		OM4123	83C451PC(M) POD-C451B(N)	OM4124: PLCC to DIL OM4125: DIL to PLCC
87C524	16K EPROM	512	20	DIL40/LCC44	UART, 3 timers Watchdog timer Bit I ² C		OM4111 + OM4110	83528PC(M) POD-C528(N)	OM4110: gen. probe OM4111: probe head
83C528 87C528	32k ROM 32k EPROM	512 512	16 16, 20	DIL40/LCC44 (QFP44)	UART, 3 timers Watchdog timer Bit I ² C		OM4111 + OM4110	83C528PC(M) POD-C528(N)	OM4110: gen. probe OM4111: probe head OM4120-S for max. speed
83CE528	32kROM	512	16	CE ONLY QFP					
83C550 87C550	4k ROM 4k EPROM	128 128	16 16	LCC44 DIL40	UART, 2 timers 8 8-bit ADC inputs, watchdog timer		OM5055 + OM4110	83550(M) POD-C550(N)	OM4110: probe base
80C552 83C552 87C552	0 8k ROM 8k EPROM	256 256 256	16, 24 16, 24 16	LCC68/QFP80	UART, 2 timers Timer with compare and capture, 2 PWM outputs, 8 10-bit ADC inputs, Byte I ² C		OM1092 + OM1095	83C552PC(M) POD-C552B(N)	OM1092: Universal probe OM1095: Upgrade unit
83CE558 87CE558 80CE558	32K ROM 32K FLASH 0	1K 1K	16 16	QFP80	As 8x552 with PLL-oscillator Auto scan ADC	89C: Q4-92 83C: Q2/3-93	OM4110 + OM4271 (in dev)		OM4110: probe base OM4115; QFP80 adapter
80C562 83C562	0 8k ROM	256 256	16 16	LCC68/QFP80	UART, 2 timers Timer with compare and capture, 2 PWM outputs, 8 8-bit ADC inputs		OM1092 + OM1095	83C552PC(M) POD-C552B(N)	OM1092: Universal probe OM1095: Upgrade unit
80C575 83C575 87C575	0 8k 8k EPROM	256 256 256	16 16 16	DIL40, LCC44 QFP44	3 timers 1 Enhanced UART, PCA, 4 analog comparators				

M = Metlink
N = Nohau

CMOS and NMOS 8-bit microcontroller family

80C51 FAMILY CMOS (Continued)

TYPE	ROM/ EPROM	RAM	SPEED (MHz)	PACKAGE	FUNCTIONS	REMARKS	PROBE SDS	THIRD PARTY EMULATOR	REMARKS
80C592 83C592 87C592	0 16k ROM 16k EPROM	512 512 512	16 16 16	LCC68/QFP80	8XC552 + CAN interface		OM4110 + OM4112	POD-592(N)	OM4110: gen. probe OM4112: probe head OM4120S: full speed
87CE598 87CE598 80CE598	32K ROM 32K EPROM 0	512 512 512	16 16 16	QFP80	8xC552 + CAN interface No I ² C	80/83CE: samp: Q1-93 prod: Q3-93 87CE: prod: Q1-93	OM4110 + OM4114		OM4110: probe base OM4115: QFP adapter
80C652 83C652 87C652	0 8k ROM 8k EPROM	256 256 256	16, 24 16, 24 16, 20	DIL40/LCC44 QFP44	UART, 2 timers Byte I ² C		OM1092 + OM1096	83652PC(M) POD-C51B(N)	
83C654 87C654	16k ROM 16k EPROM	256 256	16, 24 16, 20	DIL40/LCC44 QFP44	UART, 2 timers Byte I ² C		OM1092 + OM1096	83654(M) POD-C51B(N)	OM1092: Universal probe OM1095: Upgrade unit
83CE654	16k ROM	256	16	QFP44	UART, 2 timers Byte I ² C	83C654 with Electromagnet ic Compatibility improvements	OM1092 + OM1096	83654(M) POD-C51B(N)	OM1092: Universal probe OM1095: Upgrade unit
83C751 87C751	2k ROM 2k EPROM	64 64	16 16	DIP24 skinny LCC28 DIP24 skinny	1 timer Bit I ² C		OM1094P	83751PC(M) POD-C751(N)	
83C752	2k ROM	64	16	DIP28, LCC28	1 timer, PWM output, 5 8-bit ADC inputs, Bit I ² C		OM5072	83752A(M) POD-C752(N)	
83C752	2k EPROM	64	16	DIP 28, LCC28	1 timer, PWM output, 5 8-bit ADC inputs, Bit I ² C				
80C851 83C851	0 4k ROM	128 128	16 16	DIL40/LCC44 QFP44	UART, 2 timers 256 byte		OM1092	80851PC(M) POD-C51(N)	
83C852	6k ROM	256	6		2k byte EEPROM smart card hardware CU		OM4119		
83C053	8k ROM	192	12	DIP42 Shrunk	2 timers, 14-bit PWM, 8-6 bit PWM 128 char. OSD 3 4-bit A/D inp.		OM5054	80C053PC(M) POD-054(N)	
83C054 87C054	16k ROM 16k EPROM	192 192	12 12	DIP42 Shrunk DIP42 Shrunk	As 8XC053		OM5054	POD-054(N)	
83C055 87C055	16k ROM 16k EPROM	256 256	12 12	DIP42 Shrunk DIP42 Shrunk	As 8XC053	In dev.	OM5054		

* The following microcontrollers have no external memory access: 8XC751, 8XC752, 8XC053, 87C054, 83C852.

M = Metlink
N = Nohau

CMOS and NMOS 8-bit microcontroller family

80CLXXX FAMILY CMOS

80CLXXX FAMILY NMOS

TYPE	ROM	RAM	SPEED (MHz)	PACKAGE	FUNCTIONS	REMARKS	PROBE SDS	REMARKS
85CL000	0	256	12	Piggyback	Piggyback CL410, CL411, CL51, P80C51			
85CL580	0	256	12	Piggyback	Piggyback CL580	Q4/92		
85CL781	0	256	12	Piggyback	Piggyback CL781, CL782, CL52	Q4/92		
80CL51 80CL31	4K 0	128 128	12 12	DIL40 VSO40	2 timers, UART		OM1079	
80CL52 80CL32	8K 0	256 256	12 12	DIL40/ QFP44	3 timers, UART	Q1, 93	OM1079 + OM5004 + tbd	OM1079: Probe base OM5004: Probe adap
83CL410 80CL410	4k 0	128 128	12 12	DIL40 VSO40	2 timers Byte I ² C		OM1079	
83CL411	4k	256	12	DIL40/ QFP44	2 timers UART	Q1, 93	OM1079	
83CL580	6k	256	16	QFP64/ VSO56	3timers, UART Watchdog timer Byte I ² C, 1 PWM 4*8 bit ADC	Q4/92	OM1079 + OM5004	OM1079: Probe base OM5004: Probe adap
83CL781 83CL782	16k 16k	256 256	12 @ 4.5V 12 @ 3V	DIL40 QFP44	3timers, UART Byte I ² C	Q4/92	OM1079 + OM5004 + tbd	OM1079: Probe base OM5004: Probe adap
83CL167 83CL267	16K 12K	256 256	12 12	SDIL64 QFP64	3timers 1-14 bit PWM 4-6 bit PWM 4-7 bit PWM 4*4 bit ADC Byte I ² C 160 char OSD 126 char fonts 4 char sizes Shadow modes ODS PLL osc. 10MHz Blinking	In Dev	OM4840 OM1079	
83CL168 83CL268	16K 12K	256 256	12 12	SDIL64 QFP64	3timers 1-14 bit PWM 4-6 bit PWM 4-7 bit PWM 4*4 bit ADC RC preprocessor Byte I ² C 3 wire serial I/O 160 char OSD 126 char fonts 4 char sizes Shadow modes ODS PLL osc. 10MHz Blinking	In Dev	OM4840 + OM1079	

CMOS and NMOS 8-bit microcontroller family

8051 FAMILY NMOS

TYPE	ROM	RAM	SPEED (MHz)	PACKAGE	FUNCTIONS	REMARKS	PROBE SDS	THIRD PARTY EMULATOR	REMARKS
8051 8031	4k 0	128 128	15 15	DIL40/PLCC44 DIL40/PLCC44	UART, 2 timers		OM1091 + OM1097	8052PC(M) OPD-C51B(N)	
8052 8032	8k 0	256 256	15 15	DIL40/PLCC44 DIL40/PLCC44	UART, 3 timers UART, 3 timers		OM4111 + OM4110	8052PC(M) OPD-C51B(N)	

8048 FAMILY NMOS

TYPE	ROM	RAM	SPEED (MHz)	PACKAGE
8048 8035	1k 0	64 64	11 11	DIL40/PLCC44 DIL40/PLCC44
8049 8039	2k 0	128 128	11 11	DIL40/PLCC44 DIL40/PLCC44
8050 8040	4k 0	256 256	11 11	DIL40/PLCC44 DIL40/PLCC44

8048 FAMILY CMOS

TYPE	ROM	RAM	SPEED (MHz)	PACKAGE
80C49 80C39	2k 0	128 128	15 15	DIL40/PLCC44 DIL40/PLCC44

80C38	1k	128	15	DIL40/PLCC44	3 timers, UART Watchdog timer 8-bit I/O 1 PWM 4.5 bit ADC	OPB40 VSO8	15	528	1K	80C38
80C37	1k	128	15	DIL40/PLCC44	3 timers, UART 8-bit I/O	OPB40 VSO8	15	528	1K	80C37
80C36	1k	128	15	DIL40/PLCC44	3 timers, UART 8-bit I/O 1 PWM 4.5 bit ADC 8-bit I/O 160 char. I/O 128 char. I/O 4 char. I/O Shadow mode ODS PLL osc. 10MHz	OPB40 VSO8	15	528	1K	80C36
80C35	1k	128	15	DIL40/PLCC44	3 timers, UART 8-bit I/O 1 PWM 4.5 bit ADC 8-bit I/O 160 char. I/O 128 char. I/O 4 char. I/O Shadow mode ODS PLL osc. 10MHz	OPB40 VSO8	15	528	1K	80C35

CMOS and NMOS 8-bit microcontroller family

8400 FAMILY CMOS

TYPE	ROM	RAM	SPEED (MHz)	PACKAGE	FUNCTIONS	REMARKS	PROBE SDS	REMARKS
84C21A 84C41A 84C81A	2k 4k 8k	64 128 256	10 10 10	DIL28/SO28 DIL28/SO28 DIL28/SO28	20 I/O lines 8-bit timer Byte I ² C		OM1083	OM1025 (LSDS)
84C22A 84C42A 84C12A	2k 4k 1k	64 64 64	10 10 16	DIL20/SO20 DIL20/SO20 DIL20/SO20	13 I/O lines 8-bit timer		OM1083 + Adapter_1	OM1025 (LSDS)
84C00B	0	256	10	28 pins	20 I/O lines 8-bit timer Byte I ² C	Piggyback	OM1080	
84C00T	0	256	10	VSO-56		ROMless	OM1080	
84C121	1k	64	10	DIL20/SO20	13 I/O lines 2 8-bit timers 8 bytes EEPROM		OM1073	OM1025(LEDs)
84C121B	0	64	10			Piggyback		OM1027
84C122A 84C122B 84C422A 84C422B 84C822A 84C822B 84C822C	1k 4K 8K	32 32 32	10	A: SO20 B: SO24 C: SO28	Controller for remote control A: 12 I/O B: 16 I/O C: 20 I/O	422/822 in dev.	OM4830	
84C230	2l	64	10	DIL40/VSO40	12 I/O lines 8-bit timer 16*4 LCD drive		OM1072	
84C430	4k	128	10	QFP64	24 I/O lines 8-bit timer Byte I ² C 24*4 LCD drive		OM1072	
84C430BH	0	128	10			Piggyback for C230 and C430		
84C633	6k	256	16	VSO56	28 I/O lines 8-bit timer 16-bit up/down counter 16-bit timer with compare and capture 16*4 LCD drive		OM1086	
84C633B	0	256	16					
84C440 84C441 84C443 84C444 84C640 84C641 84C643 84C644 84C840 84C841 84C843 84C844	4k 4k 4k 4k 6k 6k 6k 6k 8k 8k 8k 8k	128 128 128 128 128 128 128 128 192 192 192 192	10 10 10 10 10 10 10 10 10 10 10 10	DIP42 shrunk	RC: 29 I/O lines LC: 28 I/O lines 8-bit timer 1 14-bit PWM 5 6-bit PWM 3-bit ADC OSD 2L-16	I ² C, RC LC I ² C, RC LC I ² C, RC LC I ² C, RC LC I ² C, RC LC I ² C, RC LC I ² C, RC LC I ² C, RC LC I ² C, RC LC I ² C, RC LC I ² C, RC LC I ² C, RC LC	OM1074	For emulation of LC versions, use OM1074 + adapter_3 + 2 adapter_5 Baud for LCDs OM4831

CMOS and NMOS 8-bit microcontroller family

8400 FAMILY CMOS (Continued)

8400 FAMILY CMOS

TYPE	ROM	RAM	SPEED (MHz)	PACKAGE	FUNCTIONS	REMARKS	PROBE SDS	REMARKS
84C646 84C846	6k 8k	192 192	10 10	DIP42 shrunk	30 I/O lines DOS clock = PLL 8 bit timer 1-14 bit PWM 4-6 bit PWM 4-7 bit PWM 3-4 bit ADC DOS: 64 disp. RAM 62 char. fonts Char. blinking Shadow modes 8 foreground colors/char. 8 background colors/word DOS: clock: 8...20MHz	I ² C, RC I ² C, RC	OM4829 + OM4832	OM4833 for LCD584
84C85 84C85B 84C853 84C853B	8k 0 8k 0	256 256 256 256	10 10 16 16	DIL40/VSO40 DIL40/VSO40 DIL40/VSO40 DIL40/VSO40	32 I/O lines 8-bit timer Byte I ² C 33 I/O lines 8-bit timer 16-bit up/down counter 16-bit timer with compare and capture	Piggyback for C85	OM1070	
84C270 84C470 84C270B 84C470B	2k 4k 0 0	128 128 128 128	10 10 10 10	DIL40/VSO40 DIL40/VSO40 DIL40/VSO40 DIL40/VSO40	8 I/O lines 16*8 capture keyboard matrix 8-bit timer 470 also handles mech. keys	Piggyback for C270 Piggyback for C470	OM1077	
84C271	2k	128	10	DIL40	8 I/O lines 16*8 mech. keyboard matrix 8-bit timer		OM1078	

8400 FAMILY NMOS

TYPE	ROM	RAM	SPEED (MHz)	PACKAGE	FUNCTIONS	REMARKS	EMULATOR TOOLS	REMARKS
8411 8421 8441 8461	1k 2k 4k 6k	64 64 128 128	6 6 6 6	DIL28/SO28 DIL28/SO28 DIL28/SO28 DIL28/SO28	20 I/O lines 8-bit timer Byte I ² C		OM1084	OM1025 (LCDS) + OM1026
8422 8442	2k 4k	64 128	6 6	DIL20 DIL20	13 I/O lines 8-bit timer Bit I ² C		PM8327/20 + PM8447	On PMDS
8401B 8401WP	0 0	128 128	6 6	28-pin PLCC68		Piggyback for 84X1 Bond out		

CMOS and NMOS 8-bit microcontroller family

3300 FAMILY CMOS

TYPE	ROM	RAM	SPEED (MHz)	PACKAGE	FUNCTIONS	REMARKS	PROBE SDS	REMARKS
3315A	1.5k	160	10	DIL28/SO28	20 I/O lines 8-bit timer $V_{DD} > 1.8V$		OM1083	OM1025(LCDS)
3343	3k	224	10	DIL28/SO28	20 I/O lines 8-bit timer $V_{DD} > 1.8V$ Byte I ² C		OM1083	OM1025(LCDS)
3344A	2k	224	3.58	DIL28/SO28	20 I/O lines 8-bit timer DTMF generator		OM1071	OM1025(LCDS) + OM1028
3346A	4k	128	10	DIL28/SO28	20 I/O lines 8-bit timer Byte I ² C 256 bytes EEPROM $V_{DD} < 1.8V$		OM1076	
3347	1.5k	64	3.58	DIL20/SO20	12 I/O lines 8-bit timer DTMF generator		OM1071 + Adapter_2	OM1025(LCDS) + OM1028
3348A	8k	256	10	DIL28/SO28	20 I/O lines 8-bit timer Byte I ² C $V_{DD} < 1.8V$		OM1083	OM1025(LCDS)
3349A	4k	224	3.58	DIL28/SO28	20 I/O lines 8-bit timer DTMF generator		OM1071	OM1025(LCDS) + OM1028
3350A	8k	128	3.58	VSO64	30 I/O lines 8-bit timer DTMF generator 256 bytes EEPROM			
3351A	2k	64	3.58	DIL28/SO28	20 I/O lines 8-bit timer DTMF generator 128 bytes EEPROM		OM5000	
3352A	6k	128	3.58	DIL28/SO28	20 I/O lines 8-bit timer DTMF generator 128 byte EEPROM		OM5000	
3353A	6k	128	16	DIL28/SO28	20 I/O lines 8-bit timer DTMF generator Ringer out 128 bytes EEPROM	March '92	OM5000	
3354A	8k	256	16	QFP64	36 I/O lines 8-bit timer DTMF generator Ringer out 256 bytes EEPROM	June '92	OM4829 + OM5003	OM4829: Probe base
3301B						Piggyback for 3315, 3343, 3348	OM1083	
3344B						Piggyback for 3344, 3347, 3349	OM1071	
3346B						Piggyback for 3346	OM1076	

(Faint bleed-through from reverse side)

[illegible]

8-bit integer

Section 1

I²C Serial Bus Application Notes & Articles

INDEX

AN422	Using the 8XC751 microcontroller as an I ² C bus master	1-3
AN425	Interfacing the PCD8584 I ² C-bus controller to 80C51 family microcontrollers	1-21
AN430	Using the 8XC751/752 in multimaster I ² C applications	1-41
AN433	I ² C slave routines for the 83C751	1-77
AN434	Connecting a PC keyboard to the I ² C-bus	1-83
AN435	Multimaster I ² C routines for byte-oriented I ² C interfaces	1-101
AN438	I ² C routines for 8XC528	1-134
AN444	Using the P82B715 I ² C extender on long cables	1-171
ETV/AN89004	PLM51 I ² C software interface IIC51 (version 0.5)	1-194
EIE/AN91007	I ² C driver routines for 8XC751/2 microcontrollers	1-210
Exploring I ² C		1-276
Programming the I ² C Interface		1-280

Using the 8XC751 microcontroller as an I²C bus master

AN422

DESCRIPTION

The 83C751/87C751 Microcontroller offers the advantages of the 80C51 architecture in a small package and at a low cost. It combines the benefits of a high-performance microcontroller with on-board hardware supporting the Inter-Integrated Circuit (I²C) bus interface.

The I²C bus, developed and patented by Philips, allows integrated circuits to communicate directly with each other via a simple bidirectional 2-wire bus. The comprehensive family of CMOS and bipolar ICs incorporating the on-chip I²C interface offers many advantages to designers of digital control for industrial, consumer and telecommunications equipment. A typical system configuration is shown in Figure 1.

Interfacing the devices in an I²C based system is very simple because they connect directly to the two bus lines: a serial data line (SDA) and a serial clock line (SCL). System design can rapidly progress from block diagram to final schematic, as there is no need to design bus interfaces, and functional blocks on a block diagram correspond to actual ICs. A prototype system or a final product version can easily be modified or upgraded by 'clipping' or 'unclipping' ICs to or from the bus. The simplicity of designing with the I²C bus does not reduce its effectiveness; it is a reliable, multimaster bus with integrated addressing and data-transfer protocols (see Figure 2). In addition, the I²C-bus compatible ICs provide cost reduction benefits to equipment manufacturers, some of which are smaller IC packages and a minimization of PCB traces and glue logic.

The availability of microcontrollers like the 83C751, with on-board I²C interface, is a very

powerful tool for system designers. The integrated protocols allow systems to be completely software defined. Software development time of different products can be reduced by assembling a library of reusable software modules. In addition, the multimaster capability allows rapid testing and alignment of end-products via external connections to an assembly-line computer.

The mask programmable 83C751 and its EPROM version, the 87C751, can operate as a master or a slave device on the I²C small area network. In addition to the efficient interface to the dedicated function ICs in the I²C family, the on-board interface facilities I/O and RAM expansion, access to EEPROM and processor-to-processor communications.

The multimaster capability of the I²C is very important but many designs do not require it. For many systems, it is sufficient that all communications between devices are initiated by a single, master processor. In this application note, use of the 8XC751 as an I²C bus master is described. Some of the technical features of the bus and the 83C751's special hardware associated with the I²C are discussed. Also included is a software example demonstrating I²C single master communications. Note that the sample routines are quite general, and therefore may be transferred easily to many applications.

The discussion of the I²C bus characteristics in this application note is by no means complete. Additional information for the I²C bus and the S83C751 Microcontroller can be found in the Microcontroller Users' Guide.

THE I²C BUS

The two lines of the I²C-bus are a serial data line (SDA) and a serial clock line (SCL). Both lines are connected to a positive supply via a pull-up resistor, and remain HIGH when the bus is not busy. Each device is recognized by a unique address—whether it is a microcomputer, LCD driver, memory or keyboard interface—and can operate as either a transmitter or receiver, depending on the function of the device. A device generating a message or data is a transmitter, and a device receiving the message or data is a receiver. Obviously, a passive function like an LCD driver could only be a receiver, while a microcontroller or a memory can both transmit and receive data.

Masters and Slaves

When a data transfer takes place on the bus, a device can either be a master or a slave. The device which initiates the transfer, and generates the clock signals for this transfer, is the master. At that time any device addressed is considered a slave. It is important to note that a master could either be a transmitter or a receiver; a master microcontroller may send data to a RAM acting as a transmitter, and then interrogate the RAM for its contents acting as a receiver—in both cases performing as the master initiating the transfer. In the same manner, a slave could be both a receiver and a transmitter.

The I²C is a multimaster bus. It is possible to have, in one system, more than one device capable of initiating transfers and controlling the bus (Figure 2). A microcontroller may act as a master for one transfer, and then be the slave for another transfer, initiated by another processor on the network. The master/slave relationships on the bus are not permanent, and may change on each transfer.

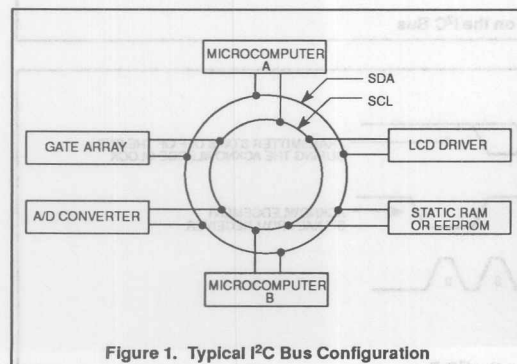


Figure 1. Typical I²C Bus Configuration

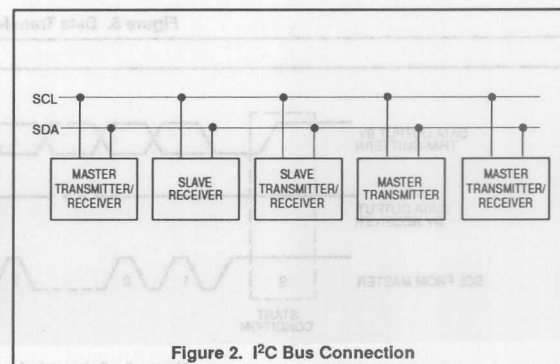


Figure 2. I²C Bus Connection

Using the 8XC751 microcontroller as an I²C bus master

AN422

As more than one master may be connected to the bus, it is possible that two devices will try to initiate a transfer at the same time. Obviously, in order to eliminate bus collisions and communications chaos, an arbitration procedure is necessary. The I²C design has an inherent arbitration and clock synchronization procedure relying on the wired-AND connection of the devices on the bus. In a typical multimaster system, a microcontroller program should allow it to gracefully switch between master and slave modes and preserve data integrity upon loss of arbitration. In this note, a simple case is presented describing the S83C751 operating as a single master on the bus.

Data Transfers

One data bit is transferred during each clock pulse (see Figure 3). The data on the SDA line must remain stable during the HIGH period of the clock pulse in order to be valid. Changes in the data line at this time will be interpreted as control signals. A HIGH-to-LOW transition of the data line (SDA) while the clock signal (SCL) is HIGH indicates a Start condition, and a LOW-to-HIGH transition of the SDA while SCL is HIGH defines a Stop condition (see Figure 4). The bus is considered to be busy after the Start condition and free again at a certain time interval after the Stop condition.

The Start and Stop conditions are always generated by the master.

The number of data bytes transferred between the Start and Stop condition from transmitter to receiver is not limited. Each byte, which must be eight bits long, is transferred serially with the most significant bit first, and is followed by an acknowledge bit. (see Figure 5). The clock pulse related to the acknowledge bit is generated by the master. The device that acknowledges has to pull down the SDA line during the acknowledge clock pulse, while the transmitting device releases the SDA line (HIGH) during this pulse (see Figure 6).

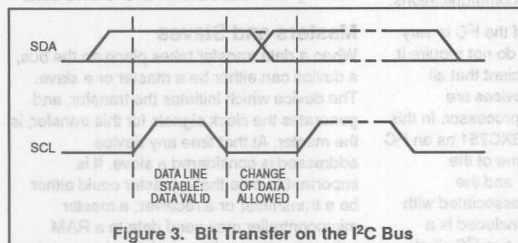
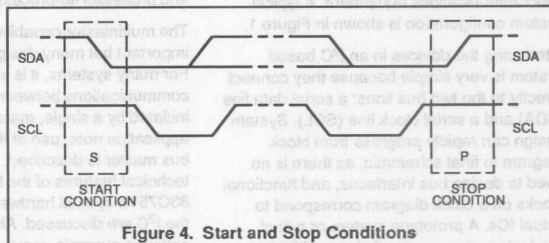
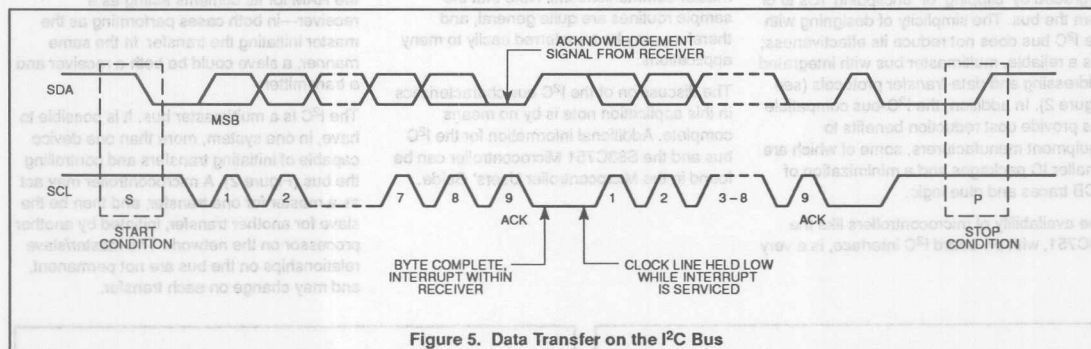
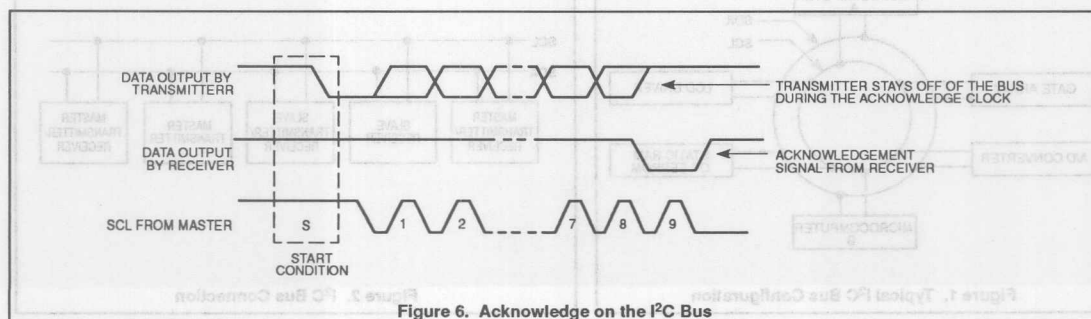
Figure 3. Bit Transfer on the I²C Bus

Figure 4. Start and Stop Conditions

Figure 5. Data Transfer on the I²C BusFigure 6. Acknowledge on the I²C Bus

Using the 8XC751 microcontroller as an I²C bus master

AN422

A slave receiver must generate an acknowledge after the reception of each byte, and a master must generate one after the reception of each byte clocked out of the slave transmitter. If a receiving device cannot receive the data byte immediately, it can force the transmitter into a wait state by holding the clock line (SCL) LOW. When designing a system, it is necessary to take into account cases when acknowledge is not received. This happens, for example, when the addressed device is busy in a real time operation. In such a case the master, after an appropriate "time-out", should abort the transfer by generating a Stop condition, allowing other transfers to take place. These "other transfers" could be initiated by other masters in a multimaster system, or by this same master.

There are two exceptions to the "acknowledge after every byte" rule. The first occurs when a master is a receiver: it must signal an end of data to the transmitter by NOT signalling an acknowledge on the last byte that has been clocked out of the slave. The acknowledge related clock, generated by the master should still take place, but the SDA line will not be pulled down. In order to indicate that this is an active and intentional lack of acknowledgement, we shall term this special condition as a "negative acknowledge".

The second exception is that a slave will send a negative acknowledge when it can no longer accept additional data bytes. This occurs after an attempted transfer that cannot be accepted.

The bus design includes special provisions for interfacing to microprocessors which implement all of the I²C communications in

software only—it is called "Slow Mode".

When all of the devices on the network have built-in I²C hardware support, the Slow Mode is irrelevant.

Addressing and Transfer Formats

Each device on the bus has its own unique address. Before any data is transmitted on the bus, the master transmits on the bus the address of the slave to be accessed for this transaction. A well-behaved slave with a matching address, if it exists on the network, should of course acknowledge the master's addressing. The addressing is done by the first byte transmitted by the master after the Start condition.

An address on the network is seven bits long, appearing as the most significant bits of the address byte. The last bit is a direction (R/W) bit. A zero indicates that the master is transmitting (WRITE) and a one indicates that the master requests data (READ). A complete data transfer, comprised of an address byte indicating a WRITE and two data bytes is shown in Figure 7.

When an address is sent, each device in the system compares the first seven bits after the Start with its own address. If there is a match, the device will consider itself addressed by the master, and will send an acknowledge. The device could also determine if in this transaction it is assigned the role of a slave receiver or slave transmitter, depending on the R/W bit.

Each node of the I²C network has a unique seven bit address. The address of a microcontroller is of course fully programmable, while peripheral devices usually have fixed and programmable address portions. In addition to the

"standard" addressing discussed here, the I²C bus protocol allows for "general call" addressing and interfacing to CBUS devices.

When the master is communicating with one device only, data transfers follow the format of Figure 7, where the R/W bit could indicate either direction. After completing the transfer and issuing a Stop condition, if a master would like to address some other device on the network, it could of course start another transaction, issuing a new Start.

Another way for a master to communicate with several different devices would be by using a "repeated start". After the last byte of the transaction was transferred, including its acknowledge (or negative acknowledge), the master issues another Start, followed by address byte and data—without effecting a Stop. The master may communicate with a number of different devices, combining READS and WRITES. After the last transfer takes place, the master issues a Stop and releases the bus. Possible data formats are demonstrated in Figure 8. Note that the repeated start allows for both change of a slave and a change of direction, without releasing the bus. We shall see later on that the change of direction feature can come in handy even when dealing with a single device.

In a single master system, the repeated start mechanism may be more efficient than terminating each transfer with a Stop and starting again. In a multimaster environment, the determination of which format is more efficient could be more complicated, as when a master is using repeated starts it occupies the bus for a long time and thus preventing other devices from initiating transfers.

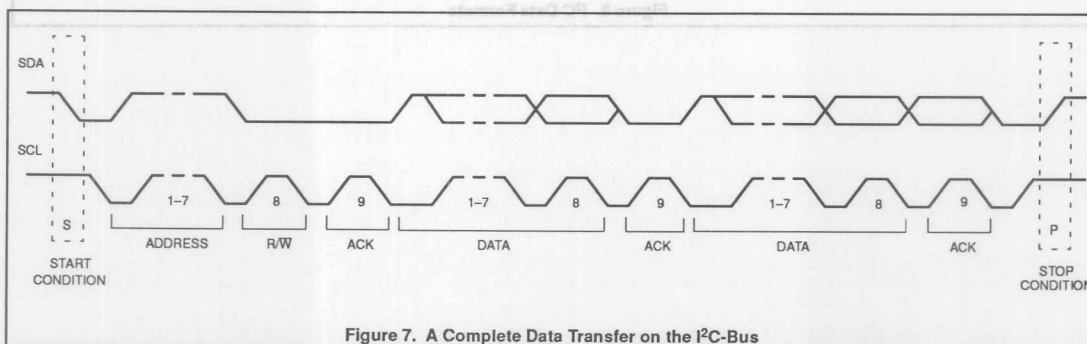


Figure 7. A Complete Data Transfer on the I²C-Bus

Using the 8XC751 microcontroller as an I²C bus master

AN422

Use of Sub-Addresses

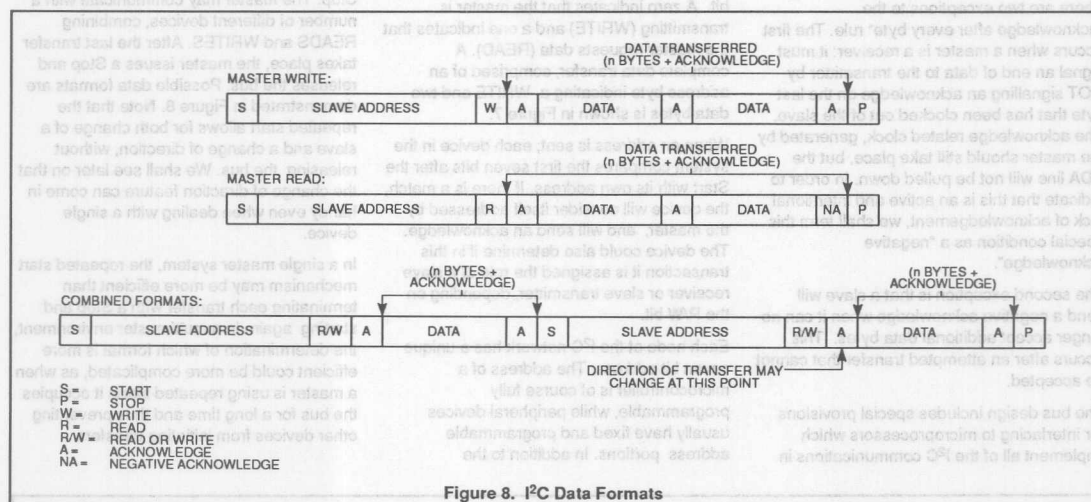
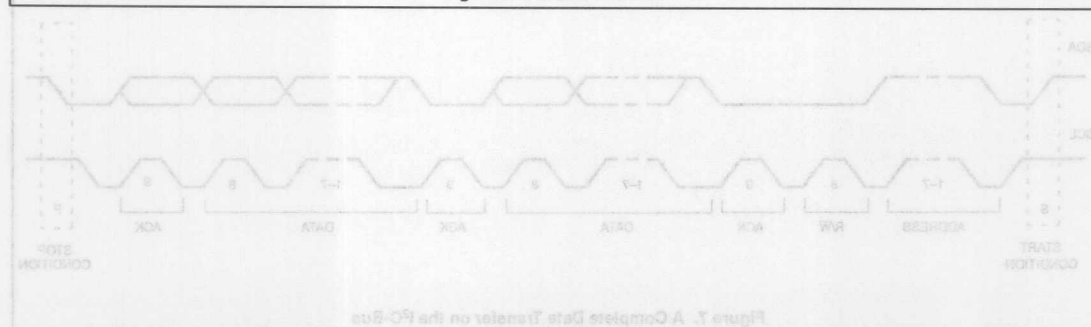
For some ICs on the I²C bus, the device address alone is not sufficient for effective communications, and a mechanism for addressing the internals of the device is necessary. A typical example when we want to access a specific word inside the device is addressing memories, or a sequence of memory locations starting at a specific internal address.

A typical I²C memory device like the PCF8570 RAM contains a built-in word address register that is incremented automatically after each data byte which is a read or written data byte. When a master communicates with the PCF8570 it must send a sub-address in the byte following the slave address byte. This sub-address is the

internal address of the word the master wants to access for a single byte transfer, or the beginning of a sequence of locations for a multi-byte transfer. A sub-address is an 8-bit byte, unlike the device address, it does not contain a direction (R/W) bit, and like any byte transferred on the bus it must be followed by an acknowledge.

A memory write cycle is shown in Figure 9(a). The Start is followed by a slave byte with the direction bit set to WRITE, a sub-address byte, a number of data bytes and a Stop signal. The sub-address is loaded into the word address memory, and the data bytes which follow will be written one after the other starting with the sub-address location, as the register is incremented automatically.

The memory read cycle (see Figure 9(b)) commences in a similar manner, with the master sending a slave address with the direction bit set to WRITE with a following sub-address. Then, in order to reverse the direction of the transfer, the master issues a repeated Start followed again by the memory device address, but this time with the direction bit set to READ. The data bytes starting at the internal sub-address will be clocked out of the device, each followed by a master-generated acknowledge. The last byte of the read cycle will be followed by a negative acknowledge, signalling the end of transfer. The cycle is terminated by a Stop signal.

Figure 8. I²C Data Formats

Using the 8XC751 microcontroller as an I²C bus master

AN422

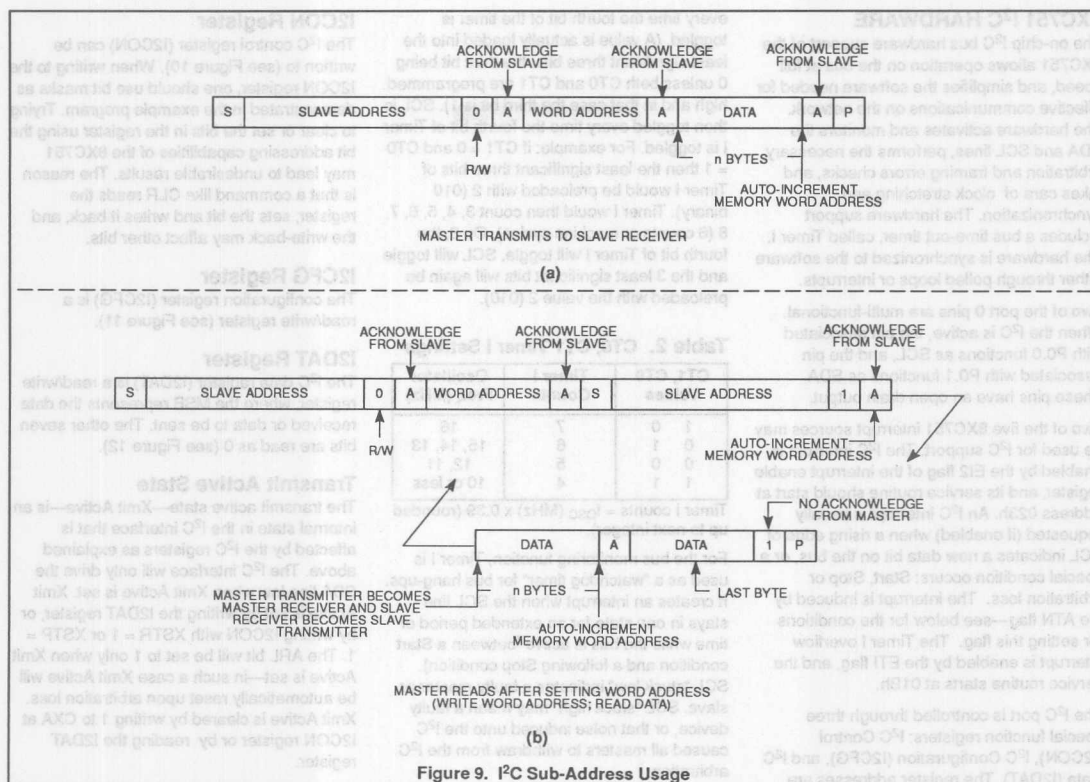
Figure 9. I²C Sub-Address Usage

Table 1. I²C Special Function Register Addresses

REGISTER			BIT ADDRESS							
Name	Symbol	Address	MSB							LSB
I ² C Control	I2CON	98	9F	9E	9D	9C	9B	9A	99	98
I ² C Data	I2DAT	99	—	—	—	—	—	—	—	—
I ² C Configuration	I2CFG	D8	DF	DE	DD	DC	DB	DA	D9	D8

Using the 8XC751 microcontroller as an I²C bus master

AN422

8XC751 I²C HARDWARE

The on-chip I²C bus hardware support of the 8XC751 allows operation on the bus at full speed, and simplifies the software needed for effective communications on the network. The hardware activates and monitors the SDA and SCL lines, performs the necessary arbitration and framing errors checks, and takes care of clock stretching and synchronization. The hardware support includes a bus time-out timer, called Timer I. The hardware is synchronized to the software either through polled loops or interrupts.

Two of the port 0 pins are multi-functional. When the I²C is active, the pin associated with P0.0 functions as SCL, and the pin associated with P0.1 functions as SDA. These pins have an open drain output.

Two of the five 8XC751 interrupt sources may be used for I²C support. The I²C interrupt is enabled by the EI2 flag of the interrupt enable register, and its service routine should start at address 023h. An I²C interrupt is usually requested (if enabled) when a rising edge of SCL indicates a new data bit on the bus, or a special condition occurs: Start, Stop or arbitration loss. The interrupt is induced by the ATN flag—see below for the conditions for setting this flag. The Timer I overflow interrupt is enabled by the ETI flag, and the service routine starts at 01Bh.

The I²C port is controlled through three special function registers: I²C Control (I2CON), I²C Configuration (I2CFG), and I²C Data (I2DAT). The register addresses are shown in Table 1.

Although the following discussion of the hardware and register details is not complete, it should give a better understanding of the programming examples.

Timer I

In I²C applications, Timer I is dedicated to the port timing generation and bus monitoring. In non-I²C applications, it is available for use as a fixed time base.

In its port timing generation function, Timer I is used to generate SCL, the I²C clock. Timer I is clocked once per machine cycle ($osc/12$), so that the toggle rate of SCL will be some multiple of that rate. Because the 83C751 can be run over a wide range of oscillator frequencies, it is necessary to adjust SCL for the part's oscillator frequency. This allows the I²C bus to be used at its highest transfer rates independent of the oscillator frequency. SCL is adjusted by writing to two bits (CT0 and CT1) in the I2CFG special function register (see Table 2). The inverse of the values in CT0 and CT1 are loaded into the least significant two bit locations of Timer I

every time the fourth bit of the timer is toggled. (A value is actually loaded into the least significant three bits, the third bit being 0 unless both CT0 and CT1 are programmed high and in that case the third bit is 1). SCL is then toggled every time the fourth bit of Timer I is toggled. For example: if CT1 = 0 and CT0 = 1 then the least significant three bits of Timer I would be preloaded with 2 (010 binary). Timer I would then count 3, 4, 5, 6, 7, 8 (6 counts or machine cycles). On 8, the fourth bit of Timer I will toggle, SCL will toggle and the 3 least significant bits will again be preloaded with the value 2 (010).

Table 2. CT0, CT1 Timer I Settings

CT1, CT0 Values	Timer I Counts	Oscillator Freq (MHz)
1 0	7	16
0 1	6	15, 14, 13
0 0	5	12, 11
1 1	4	10 or less

Timer I counts = f_{osc} (MHz) \times 0.39 (rounded up to next integer).

For the bus monitoring function, Timer I is used as a "watchdog timer" for bus hang-ups. It creates an interrupt when the SCL line stays in one state for an extended period of time while the bus is active (between a Start condition and a following Stop condition). SCL "stuck low" indicates a faulty master or slave. SCL "stuck high" may mean a faulty device, or that noise induced onto the I²C caused all masters to withdraw from the I²C arbitration.

The time-out interval of Timer I is fixed (cannot be set): it carries out and interrupts (if enabled) when about 1024 machine cycles have elapsed since a change on SCL within a frame. In other words, whenever I²C is active and Timer I is enabled, the falling edge of SCL will reset Timer I. If SCL is not toggled low for 1024 machine cycles, Timer I will overflow and cause an interrupt. (Note: we wrote "about 1024 machine cycles" although for the sake of accuracy—this number is affected by the setting of the CT0 and CT1 bits mentioned above and may vary by up to three machine cycles) The exact number of cycles for a time-out is not critical; what is important is that it indicates SCL is stuck.

In addition to the interrupt, upon Timer I overflow the I²C port hardware is reset. This is useful for multiple master systems in situations where a bus fault might cause the bus to hang-up due to a lack of software response. When this happens, SCL will be released, and I²C operation between other devices can continue.

I2CON Register

The I²C control register (I2CON) can be written to (see Figure 10). When writing to the I2CON register, one should use bit masks as demonstrated in the example program. Trying to clear or set the bits in the register using the bit addressing capabilities of the 8XC751 may lead to undesirable results. The reason is that a command like CLR reads the register, sets the bit and writes it back, and the write-back may affect other bits.

I2CFG Register

The configuration register (I2CFG) is a read/write register (see Figure 11).

I2DAT Register

The I²C data register (I2DAT) is a read/write register, where the MSB represents the data received or data to be sent. The other seven bits are read as 0 (see Figure 12).

Transmit Active State

The transmit active state—Xmit Active—is an internal state in the I²C interface that is affected by the I²C registers as explained above. The I²C interface will only drive the SDA line low when Xmit Active is set. Xmit Active is set by writing the I2DAT register, or by writing I2CON with XSTR = 1 or XSTP = 1. The ARL bit will be set to 1 only when Xmit Active is set—in such a case Xmit Active will be automatically reset upon arbitration loss. Xmit Active is cleared by writing 1 to CXA at I2CON register or by reading the I2DAT register.

PROGRAMMING EXAMPLE

The listing demonstrates communications routines for the 8XC751 as an I²C bus master in a single-master system.

The single-master system is less complicated than a multimaster environment. The programmer does not have to worry about switching between master and slave roles, or the consequences of an arbitration loss.

The I²C interrupt is not used, and therefore disabled. There is no need for frame Start interrupts, as this processor is the only bus master and all data transfers are initiated by it when the appropriate routines are called by the application. No one else generates frame Starts which could be an interrupt source in a multimaster system. Within the frames we monitor bus activity with a wait-loop which polls the ATN flag. As we expect the bus to operate in its full-speed mode, we can assume that only a small amount of time will

Using the 8XC751 microcontroller as an I²C bus master

AN422

be wasted in those loops, and the use of interrupts would be less efficient.

The 8XC751 has single-bit I²C hardware interface, where the registers may directly affect the levels on the bus and the software interacting with the register takes part in the protocol implementation. The hardware and the low-level routines dealing with the registers are tightly coupled. Therefore, one should take extra care if trying to modify these lower level routines.

The beginning of the program, at address 0, contains the reset vector, where the microcontroller begins executing code after a hardware reset. In this case, the code simply jumps to the main part of the program, which begins at the label 'Reset' near the end of the listing.

The main program is a simple demonstration of the I²C routines which comprise the balance of the listing. It first enables the Timer 1 interrupt, and sets up some sample data to be transmitted. Beginning at the label MainLoop, the program then proceeds to transmit one byte of data to a slave device at address 48 hexadecimal, using the routine titled 'SendData'. In our demonstration hardware, this address corresponds to an 8-bit I/O port that drives eight monitor LEDs. The program then reads back one byte of data from the same port using the routine 'RcvData'. The SendData and RcvData routines can send or receive multiple bytes of data, the number of which is determined by the variable 'ByteCnt'.

Upon return from both SendData and RcvData, the program checks the system flag named 'Retry' to see if the transfer was completed correctly. If not, it loops back and attempts the same transfer again.

Next, the program sends four bytes of data to a 256-byte EEPROM device, an 8-pin part called the PCF8582. The routine 'SendSub' is used for this purpose. The EEPROM was located at address A0 hexadecimal on our board. This device uses the sub-addressing feature to select a starting location to address in the EEPROM array. When data is written to the EEPROM, the address is automatically incremented so that the data bytes are stored in consecutive locations.

Finally the program reads back four bytes of data from the EEPROM using the routine 'RcvSub'. Calls to SendSub and RcvSub should also be followed by a test of the Retry flag to insure that all went according to plan.

This entire process is repeated indefinitely by jumping back to MainLoop.

Back at the beginning of the program, the next location after the reset vector is the Timer 1 interrupt service routine. The microcontroller will go to address 1B hexadecimal if Timer 1 overflows. This routine stops the timer, clears the timer interrupt, clears the pending interrupt so that other interrupts will be enabled, restores the stack pointer, and jumps to the 'Recover' routine to try to correct whatever stopped the I²C bus and allowed Timer 1 to overflow.

Next in the listing come the main I²C service routines. These are the routines SendData, RcvData, SendSub, and RcvSub that were called from the main program. Both of the send routines use the data area labeled 'XmtDat' as the transmit data buffer. In this sample program, four bytes were reserved for this area, but it could be larger or smaller depending on the application. The two receive routines use another four byte buffer labeled 'RcvDat' to store received data. All of these routines use the variables 'SlvAdr' and 'ByteCnt' to determine the slave address and the number of bytes to be sent or received, respectively. The SendSub and RcvSub routines use the variable 'SubAdr' as the sub-address to send to the slave device.

Following the main I²C service routines in the listing are the subroutines that are called by the main routines to deal intimately with the I²C hardware.

The 'SendAddr' subroutine requests mastership of the I²C bus and calls the routine 'XmitAddr' to complete sending the slave address. The bulk of the XmitAddr routine is shared with the 'XmitByte' subroutine which sends data bytes on the I²C bus. XmitByte is also used to send I²C sub-addresses. Both subroutines check for an acknowledge from the slave device after every byte is sent on the I²C bus.

The next subroutine 'RDack' calls the 'RcvByte' routine to read in a byte of data. It then sends an acknowledge to the slave device. RDack is used to receive all data except for the last byte of a receive data frame, where the acknowledge is omitted by the bus master. The RcvByte subroutine is called directly for the last byte of a frame.

The 'SendStop' subroutine causes a stop condition on the I²C, thus ending a frame. The 'RepStart' subroutine sends a repeated start condition on the I²C bus, to allow the master to start a new frame without first having to send an intervening stop.

The lower level subroutines deal directly with the hardware. The tight coupling between

hardware and software is best demonstrated by the following explanations, relating to two cases in which the code is not self evident.

Sending the Address

When sending the address byte in the SendAddr subroutine, the first bit is written to I2DAT prior to the loop where the other seven bits are sent (SendAd2). The reason is that we need to clear the Start condition in order to release the SCL line, and this is done explicitly by the subsequent command. When SCL is released, the correct bit (MSB of address) must already be in I2DAT.

Capturing the Received Data

Typically, a program receiving data waits in a loop for ATN, and when detected, checks DRDY. If DRDY = 1 then there was a rising SCL, and the new data can be read from RDAT in I2CON or I2DAT. Reading or writing I2DAT clears DRDY, thus releasing SCL.

When reading the last bit in a byte, it should be read from I2CON, and not I2DAT (see the end of the RcvByte routine). This way the Data Ready (DRDY) flag is not cleared, and the low period on SCL is stretched. The reason for doing so is that upon reception of the last bit of a received byte the master must react with an acknowledge. In order to ensure that we "wait" with the acknowledge clock (release of SCL) until the acknowledge level is issued on SDA, the last bit is read out of I2CON and not I2DAT. SCL is stretched low until the acknowledge level is written into I2DAT by the software.

Bus Faults and Other Exceptions

Bus exceptions are detected either by Timer 1 time-out, or "illegal" logic states tested for and detected by the software. Upon Timer 1 time-out, a bus recovery is attempted by the Recover routine. The final section of the listing is this 'Recover' routine. Its job is to try to restore control of the I²C bus to the main program. First, the subroutine 'FixBus' is called. It checks to see if only the SDA line is 'stuck', and if so, tries to correct it by sending some extra clocks on the SCL line, and forcing a stop condition on the bus. If this does not work, another subroutine 'BusReset' is called. This generally happens when a severe bus error occurs, such as a shorted clock line. The philosophy used in this code is that the only chance of recovering from a severe error is to cause a reset of the I²C hardware by deliberately forcing Timer 1 to time out. This method allows recovery from a temporary short or other serious condition on the I²C bus.

Using the 8XC751 microcontroller as an I²C bus master

AN422

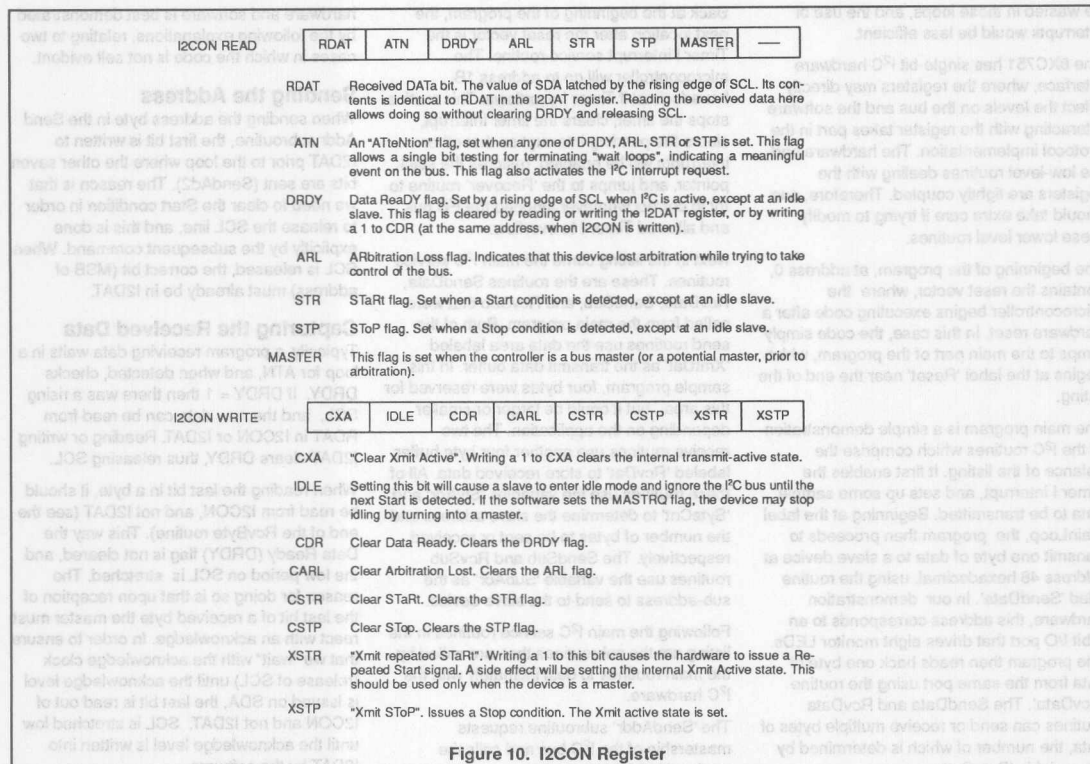


Figure 10. I2CON Register

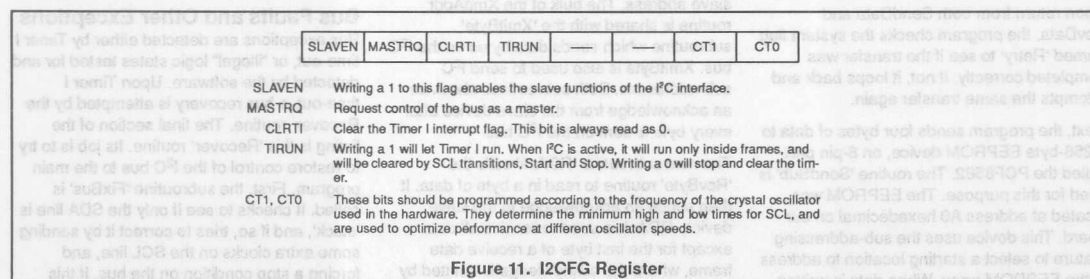


Figure 11. I2CFG Register

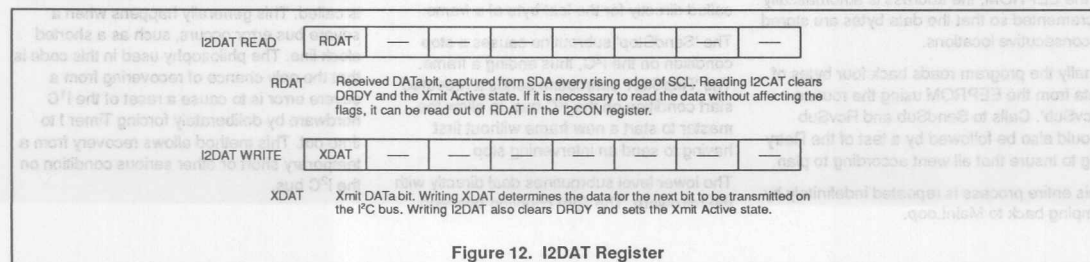


Figure 12. I2DAT Register

Using the 8XC751 microcontroller as an I²C bus master

AN422

I2CAPP	83C751 Single Master I2C Routines	09/07/89	
	1 *****		
	2 ;*****		
	3 ;		
	4 ; Sample I2C Single Master Routines for the 83C751		
	5 ;*****		
	6 ;*****		
	7		
	8 \$TITLE(83C751 Single Master I2C Routines)		
	9 \$DATE(09/07/89)		
	10 \$MOD751		
	11 \$DEBUG		
	12		
	13		
	14 ; Value definitions.		
	15		
0002	16 CTVAL EQU 02h ;CT1, CT0 bit values for I2C.		
	17		
	18		
	19 ; Masks for I2CFG bits.		
	20		
0010	21 BTIR EQU 10h ;Mask for TIRUN bit.		
0040	22 BMRQ EQU 40h ;Mask for MASTRQ bit.		
	23		
	24		
	25 ; Masks for I2CON bits.		
	26		
0080	27 BCXA EQU 80h ;Mask for CXA bit.		
0040	28 BIDL EQU 40h ;Mask for IDLE bit.		
0020	29 BCDR EQU 20h ;Mask for CDR bit.		
0010	30 BCARL EQU 10h ;Mask for CARL bit.		
0008	31 BCSTR EQU 08h ;Mask for CSTR bit.		
0004	32 BCSTP EQU 04h ;Mask for CSTP bit.		
0002	33 BXSTR EQU 02h ;Mask for XSTR bit.		
0001	34 BXSTP EQU 01h ;Mask for XSTP bit.		
	35		
	36		
	37 ; RAM locations used by I2C routines.		
	38		
0021	39 BitCnt DATA 21h ;I2C bit counter.		
0022	40 ByteCnt DATA 22h		
0023	41 SlvAdr DATA 23h ;Address of active slave.		
0024	42 SubAdr DATA 24h		
	43		
0025	44 RcvDat DATA 25h ;I2C receive data buffer (4 bytes).		
	45 ; addresses 25h through 28h.		
	46		
0029	47 XmtDat DATA 29h ;I2C transmit data buffer (4 bytes).		
	48 ; addresses 29h through 2Ch.		
	49		
002D	50 StackSave DATA 2Dh ;Saves stack addr for bus recovery.		
	51		
0020	52 Flags DATA 20h ;I2C software status flags.		
0000	53 NoAck BIT Flags.0 ;Indicates missing acknowledge.		
0001	54 Fault BIT Flags.1 ;Indicates a bus fault of some kind.		
0002	55 Retry BIT Flags.2 ;Indicates that last I2C transmission		
	56 ; failed and should be repeated.		
	57		
0080	58 SCL BIT P0.0 ;Port bit for I2C serial clock line.		
0081	59 SDA BIT P0.1 ;Port bit for I2C serial data line.		
	60		

```

61 ;*****
62 ;                               Begin Code
63 ;*****
64 ;*****
65 ; Reset and interrupt vectors.
66 ;*****
0000 21E1 67      AJMP      Reset      ;Reset vector at address 0.
68 ;*****
69 ;*****
70 ; A timer I timeout usually indicates a 'hung' bus.
71 ;*****
001B      72      ORG      1Bh      ;Timer I (I2C timeout):
; interrupt.
001B D2DD 73      TimerI:  SETB      CLRTI      ;Clear timer I interrupt.
001D C2DC 74      CLR      TIRUN      ;*****
001F 1126 75      ACALL     ClrInt      ;Clear interrupt pending.
0021 852D81 76      MOV      SP,StackSave ;Restore stack for return
; to main.
0024 218A 77      AJMP      Recover      ;Attempt bus recovery.
0026 32      78      ClrInt:  RETI
79 ;*****
80 ;*****
81 ;*****
82 ;                               Main Transmit and Receive Routines
83 ;*****
84 ;*****
85 ; Send data byte(s) to slave.
86 ; Enter with slave address in SlvAdr, data in XmtDat buffer,
87 ; # of data bytes to send in ByteCnt.
88 ;*****
0027 C200 89      SendData: CLR      NoAck      ;Clear error flags.
0029 C201 90      CLR      Fault      ;*****
002B C202 91      CLR      Retry      ;*****
002D 85812D 92      MOV      StackSave,SP ;Save stack address
; for bus fault.
0030 E523 93      MOV      A,SlvAdr      ;Get slave address.
0032 310C 94      ACALL     SendAddr      ;Get bus and send slave addr.
0034 200012 95      JB      NoAck,SDEX      ;Check for missing
; acknowledge.
0037 200112 96      JB      Fault,SDataErr ;Check for bus fault.
003A 7829 97      MOV      RO,#XmtDat ;Set start of transmit
; buffer.
98 ;*****
003C E6      99      SDLoop:  MOV      A,@RO      ;Get data for slave.
003D 08      100      INC      RO      ;*****
003E 3125 101      ACALL     XmitByte      ;Send data to slave.
0040 200006 102      JB      NoAck,SDEX      ;Check for missing
; acknowledge.
0043 200106 103      JB      Fault,SDataErr ;Check for bus fault.
0046 D522F3 104      DJNZ      ByteCnt,SDLoop ;*****
105 ;*****
0049 3166 106      SDEX:  ACALL     SendStop      ;Send an I2C stop.
004B 22      107      RET
108 ;*****
109 ;*****
110 ; Handle a transmit bus fault.
111 ;*****
004C 218A 112      SDataErr: AJMP      Recover      ;Attempt bus recovery.
113 ;*****
114 ;*****
115 ; Receive data byte(s) from slave.
116 ; Enter with slave address in SlvAdr,
; # of data bytes requested in ByteCnt.
117 ; Data returned in RcvDat buffer.

```

Using the 8XC751 microcontroller as an I²C bus master

AN422

004E C200	118				
0050 C201	119	RcvData:	CLR	NoAck	;Clear error flags.
0052 C202	120		CLR	Fault	
0054 85812D	121		CLR	Retry	
	122		MOV	StackSave,SP	;Save stack address
					; for bus fault.
0057 E523	123		MOV	A,SlvAdr	;Get slave address.
0059 D2E0	124		SETB	ACC.0	;Aet bus read bit.
005B 310C	125		ACALL	SendAddr	;Send slave address.
005D 200023	126		JB	NoAck,RDEX	;Check for missing
					; acknowledge.
0060 200123	127		JB	Fault,RDatErr	;Check for bus fault.
	128				
0063 7825	129		MOV	R0,RcvDat	;Set start of receive
					; buffer.
0065 D52202	130		DJNZ	ByteCnt,RDLoop	;Check for count = 1
					; byte only.
0068 800A	131		SJMP	RDLast	
006A 3143	132				
	133	RDLoop:	ACALL	RDACK	;Get data and send
					; an acknowledge.
006C 200117	134		JB	Fault,RDatErr	;Check for bus fault.
006F F6	135		MOV	@R0,A	;Save data.
0070 08	136		INC	R0	
0071 D522F6	137		DJNZ	ByteCnt,RDLoop	;Repeat until last
					; byte.
	138				
0074 314F	139	RDLast:	ACALL	RcvByte	;Get last data byte
					; from slave.
0076 20010D	140		JB	Fault,RDatErr	;Check for bus
					; fault.
0079 F6	141		MOV	@R0,A	;Save data.
	142				
007A 759980	143		MOV	I2DAT,#80h	;Send negative
					; acknowledge.
007D 309EFD	144		JNB	ATN,\$;Wait for NAK sent.
0080 309D03	145		JNB	RDY,RDatErr	;Check for bus
					; fault.
	146				
0083 3166	147	RDEX:	ACALL	SendStop	;Send an I2C bus
					; stop.
0085 22	148		RET		
	149				
	150				
	151				
	152				
0086 218A	153	RDatErr:	AJMP	Recover	;Attempt bus recovery.
	154				
	155				
	156				; Send data byte(s) to slave with subaddress.
	157				; Enter with slave address in ACC, subaddress in
					; SubAdr, # of bytes to send in ByteCnt,
	158				; data in XmtDat buffer.
	159				
0088 C200	160	SendSub:	CLR	NoAck	;Clear error flags.
008A C201	161		CLR	Fault	
008C C202	162		CLR	Retry	
008E 85812D	163		MOV	StackSave,SP	;Save stack address
					; for bus fault.
0091 E523	164		MOV	A,SlvAdr	;Get slave address.
0093 310C	165		ACALL	SendAddr	;Get bus and send
					; slave address.

Using the 8XC751 microcontroller as an I²C bus master

AN422

0095 20001C	166	JB	NoAck,SSEX	;Check for missing ; acknowledge.
0098 20011C	167	JB	Fault,SSubErr	; Check for bus ; fault.
	168			
009B E524	169	MOV	A,SubAdr	;Get slave subaddress.
009D 3125	170	ACALL	XmitByte	;Send subaddress.
009F 200012	171	JB	NoAck,SSEX	;Check for missing ; acknowledge.
00A2 200112	172	JB	Fault,SSubErr	;Check for bus fault.
00A5 7829	173	MOV	R0,#XmtDat	;Set start of ; transmit buffer.
	174			
00A7 E6	175	SSLoop: MOV	A,@R0	;Get data for slave.
00A8 08	176	INC	R0	
00A9 3125	177	ACALL	XmitByte	;Send data to slave.
00AB 200006	178	JB	NoAck,SSEX	;Check for missing ; acknowledge.
00AE 200106	179	JB	Fault,SSubErr	;Check for bus fault.
00B1 D522F3	180	DJNZ	ByteCnt,SSLoop	
	181			
00B4 3166	182	SSEX: ACALL	SendStop	;Send an I2C stop.
00B6 22	183	RET		
	184			
	185			
	186			; Handle a transmit bus fault.
	187			
00B7 218A	188	SSubErr: AJMP	Recover	;Attempt bus recovery.
	189			
	190			
	191			; Receive data byte(s) from slave with subaddress.
	192			; Enter with slave address in SlvAdr, subaddress in SubAdr, ; # of data bytes requested in ByteCnt.
	193			; Data returned in RcvDat buffer.
	194			
00B9 C200	195	RcvSub: CLR	NoAck	;Clear error flags.
00BB C201	196	CLR	Fault	
00BD C202	197	CLR	Retry	
00BF 85812D	198	MOV	StackSave,SP	;Save stack address ; for bus fault.
00C2 E523	199	MOV	A,SlvAdr	;Get slave address.
00C4 310C	200	ACALL	SendAddr	;Send slave address.
00C6 20003E	201	JB	NoAck,RSEX	;Check for missing ; acknowledge.
00C9 20013E	202	JB	Fault,RSubErr	;Check for bus fault.
	203			
00CC E524	204	MOV	A,SubAdr	;Get slave subaddress.
00CE 3125	205	ACALL	XmitByte	;Send subaddress.
00D0 200034	206	JB	NoAck,RSEX	;Check for missing ; acknowledge.
00D3 200134	207	JB	Fault,RSubErr	;Check for bus fault.
	208			
00D6 317A	209	ACALL	RepStart	;Send repeated start.
00D8 20012F	210	JB	Fault,RSubErr	;Check for bus fault.
00DB E523	211	MOV	A,SlvAdr	;Get slave address.
00DD D2E0	212	SETB	ACC.0	;Set bus read bit.
00DF 3115	213	ACALL	SendAd2	;Send slave address.
00E1 200023	214	JB	NoAck,RSEX	;Check for missing ; acknowledge.
00E4 200123	215	JB	Fault,RSubErr	;Check for bus fault.
	216			
00E7 7825	217	MOV	R0,#RcvDat	;Set start of ; receive buffer.
00E9 D52202	218	DJNZ	ByteCnt,RSLoop	;Check for count = 1 ; byte only.

Using the 8XC751 microcontroller as an I²C bus master

AN422

00EC 800A	219	SJMP	RSLast		
	220				
00EE 3143	221	RSLoop:	ACALL	RDACK, TAG01	;Get data and send
					; an acknowledge.
00F0 200117	222	JB	Fault, RSubErr		;Check for bus fault.
00F3 F6	223	MOV	@R0, AX		;Save data.
00F4 08	224	INC	R0		
00F5 D522F6	225	DJNZ	ByteCnt, RSLoop		;Repeat until last byte.
	226				
00F8 314F	227	RSLast:	ACALL	RcvByte, WTA	;Get last data byte
					; from slave.
00FA 20010D	228	JB	Fault, RSubErr		;Check for bus fault.
00FD F6	229	MOV	@R0, A		;Save data.
	230				
00FE 759980	231	MOV	I2DAT, #80h		;Send negative
					; acknowledge.
0101 309EFD	232	JNB	ATN, \$;Wait for NAK sent.
0104 309D03	233	JNB	DRDY, RSubErr		;Check for bus fault.
	234				
0107 3166	235	RSEX:	ACALL	SendStop	;Send an I2C bus stop.
0109 22	236	RET			
	237				
	238				
	239				; Handle a receive bus fault.
	240				
010A 218A	241	RSubErr:	AJMP	Recover	;Attempt bus recovery.
	242				
	243				
	244				;*****
	245				Subroutines
	246				;*****
	247				
	248				; Send address byte.
	249				; Enter with address in ACC.
	250				
010C 75D852	251	SendAddr:	MOV	I2CFG, #BMQ+BTIR+CTVAL	;Request I2C bus.
010F 309EFD	252	JNB	ATN, \$;Wait for bus
					; granted.
0112 309908	253	JNB	Master, SAErr		;Should have
					; become the bus
					; master.
0115 F599	254	SendAd2:	MOV	I2DAT, A	;Send first bit,
					; clears DRDY.
0117 75981C	255	MOV	I2CON, #BCARL+BCSTR+BCSTP		;Clear start,
					; releases SCL.
011A 3120	256	ACALL	XmitAddr		;Finish sending
					; address.
011C 22	257	RET			
	258				
011D D201	259	SAErr:	SETB	Fault	;Return bus fault
					; status.
011F 22	260	RET			
	261				
	262				
	263				; Byte transmit routine.
	264				; Enter with data in ACC.
	265				XmitByte : transmits 8 bits.
	266				XmitAddr : transmits 7 bits (for address only).
	267				
0120 752108	268	XmitAddr:	MOV	BitCnt, #8	;Set 7 bits of
					; address count.
0123 8005	269	SJMP	XmBit2		
	270				

Using the 8XC751 microcontroller as an I²C bus master

AN422

0125 752108	271	XmitByte: MOV	BitCnt,#8	;Set 8 bits of data ; count.
0128 F599	272	XmBit: MOV	I2DAT,A	;Send this bit.
012A 23	273	XmBit2: RL	A	;Get next bit.
012B 309EFD	274	JNB	ATN,\$;Wait for bit sent.
012E 309D0F	275	JNB	DRDY,XMErr	;Should be data ready.
0131 D521F4	276	DJNZ	BitCnt,XmBit	;Repeat until all bits sent.
0134 7598A0	277	MOV	I2CON,#BCDR+BCXA	;Switch to ; receive mode.
0137 309EFD	278	JNB	ATN,\$;Wait for acknowledge ; bit.
013A 309F02	279	JNB	RDAT,XMBX	;Was there an ack?
013D D200	280	SETB	NoAck	;Return no acknowledge ; status.
013F 22	281	XMBX: RET		
	282			
0140 D201	283	XMErr: SETB	Fault	;Return bus fault ; status.
0142 22	284	RET		
	285			
	286			
	287			; Byte receive routines.
	288			; RDack : receives a byte of data, then sends ; an acknowledge.
	289			; RcvByte : receives a byte of data.
	290			; Data returned in ACC.
	291			
0143 314F	292	RDack: ACALL	RcvByte	;Receive a data byte.
0145 759900	293	MOV	I2DAT,#0	;Send receive ; acknowledge.
0148 309EFD	294	JNB	ATN,\$;Wait for acknowledge ; sent.
014B 309D15	295	JNB	DRDY,RdErr	;Check for bus fault.
014E 22	296	RET		
	297			
014F 752108	298	RcvByte: MOV	BitCnt,#8	;Set bit count.
0152 E4	299	CLR	A	;Init received byte ; to 0.
0153 4599	300	ORL	A,I2DAT	;Get bit, clear ATN.
0155 23	301	RL	A	;Shift data.
0156 309EFD	302	JNB	ATN,\$;Wait for next bit.
0159 309D07	303	JNB	DRDY,RdErr	;Should be data ready.
015C D521F4	304	DJNZ	BitCnt,RBit	;Repeat until 7 bits ; are in.
015F A29F	305	MOV	C,RDAT	;Get last bit, don't ; clear ATN.
0161 33	306	RLC	A	;Form full data byte.
0162 22	307	RET		
	308			
0163 D201	309	RdErr: SETB	Fault	;Return bus fault status.
0165 22	310	RET		
	311			
	312			
	313			; I2C stop routine.
	314			
0166 C2DE	315	SendStop: CLR	MASTRQ	;Release bus ; mastership.
0168 759821	316	MOV	I2CON,#BCDR+BXSTP	;Generate a bus stop.
016B 309EFD	317	JNB	ATN,\$;Wait for atn.
016E 759820	318	MOV	I2CON,#BCDR	;Clear data ready.
0171 309EFD	319	JNB	ATN,\$;Wait for stop sent.
0174 759894	320	MOV	I2CON,#BCARL+BCSTP+BCXA	;Clear I2C bus.
0177 C2DC	321	CLR	TIRUN	;Stop timer I.
0179 22	322	RET		
	323			

Using the 8XC751 microcontroller as an I²C bus master

AN422

	324								
	325								
	326								
	327								
017A 759822	328	RepStart:	MOV	I2CON,#BCDR+BXSTR					
017D 309EFD	329		JNB	ATN,\$					
0180 759820	330		MOV	I2CON,#BCDR					
0183 309EFD	331		JNB	ATN,\$					
0189 22	333		RET						
	334								
	335								
	336								
	337								
018A 31A4	338	Recover:	ACALL	FixBus					
018C 400D	339		JC	BusReset					
018E D202	340		SETB	Retry					
0190 C201	341		CLR	Fault					
0192 C200	342		CLR	NoAck					
0194 D2DD	343		SETB	CLRTI					
0196 D2DC	344		SETB	TIRUN					
0198 D2AB	345		SETB	ETI					
019A 22	346		RET						
	347								
	348								
	349								
	350								
	351								
019B C2DE	352	BusReset:	CLR	MASTRQ					
019D 7598BC	353		MOV	I2CON,#0Bch					
01A0 D2DC	354		SETB	TIRUN					
01A2 80FE	355		SJMP	\$+1					
	356								
	357								
	358								
	359								
	360								
	361								
01A4 C2DE	362	FixBus:	CLR	MastrQ					
01A6 D3	363		SETB	C					
01A7 D280	364		SETB	SCL					
01A9 D281	365		SETB	SDA					
01AB 308029	366		JNB	SCL,FixBusEx					
01AE 208113	367		JB	SDA,RStoP					
01B1 752109	368		MOV	BitCnt,#9					
01B4 C280	369	ChekLoop:	CLR	SCL					
01B6 31D8	370		ACALL	SDelay					
01B8 208109	371		JB	SDA,RStoP					
01BB D280	372		SETB	SCL					
01BD 31D8	373		ACALL	SDelay					
01BF D521F2	374		DJNZ	BitCnt,ChekLoop					
	375								

Using the 8XC751 microcontroller as an I²C bus master

AN422

```

01C2 8013          376          SJMP    FixBusEx      ;Failed to fix bus by
                                ;this method.
                                377
01C4 C281          378      RStop:  CLR    SDA        ;Try forcing a stop
                                ;since SCL & SDA
01C6 31D8          379          ACALL  SDelay        ;are both high.
01C8 D280          380          SETB   SCL
01CA 31D8          381          ACALL  SDelay
01CC D281          382          SETB   SDA
01CE 31D8          383          ACALL  SDelay
01D0 308004        384          JNB    SCL,FixBusEx    ;Are SCL & SDA still
                                ;high? If so, assume bus
01D3 308101        385          JNB    SDA,FixBusEx    ;is now OK, and return
01D6 C3            386          CLR    C              ;with carry cleared.
01D7 22            387      FixBusEx: RET
                                388
                                389
                                390      ; Short delay routine (10 machine cycles).
                                391
01D8 00            392      SDelay: NOP
01D9 00            393      NOP
01DA 00            394      NOP
01DB 00            395      NOP
01DC 00            396      NOP
01DD 00            397      NOP
01DE 00            398      NOP
01DF 00            399      NOP
01E0 22            400      RET
                                401
                                402      ;*****
                                403
                                404      ; Main Program
                                405      ;*****
                                406
01E1 758107        407      Reset: MOV    SP,#07h      ;Set stack location.
01E4 D2AB          408          SETB   ETIF0,MOO0I    ;Enable timer I interrupts.
01E6 D2AF          409          SETB   EA          ;Enable global interrupts.
01E8 75290B        410          MOV    XmtDat,#11h    ;Set up transmit data.
01EB 752A16        411          MOV    XmtDat+1,#22 ;Set up transmit data.
01EE 752B2C        412          MOV    XmtDat+2,#44 ;Set up transmit data.
01F1 752C58        413          MOV    XmtDat+3,#88 ;Set up transmit data.
01F4 752500        414          MOV    RcvDat,#0      ;Clear receive data.
01F7 752600        415          MOV    RcvDat+1,#0    ;Clear receive data.
01FA 752700        416          MOV    RcvDat+2,#0    ;Clear receive data.
01FD 752800        417          MOV    RcvDat+3,#0    ;Clear receive data.
                                418
0200 752348        419      MainLoop: MOV    SlvAdr,#48h ;Set slave address
                                ; (8-bit I/O port).
0203 752201        420          MOV    ByteCnt,#10 ;Set up byte count.
0206 1127          421          ACALL  SendData    ;Send data to slave.
0208 2002F5        422          JB     Retry,MainLoop
                                423
020B 752201        424      ML2:  MOV    ByteCnt,#10 ;Set up byte count.
020E 114E          425          ACALL  RcvData     ;Read data from slave.
0210 2002F8        426          JB     Retry,ML2
                                427
0213 7523A0        428      SL1:  MOV    SlvAdr,#0A0h ;Set slave address
                                ; (RAM chip).
0216 752400        429          MOV    SubAdr,#0h    ;Set slave subaddress.
0219 752204        430          MOV    ByteCnt,#4     ;Set up byte count.
021C 1188          431          ACALL  SendSub     ;
021E 2002F2        432          JB     Retry,SL1
                                433

```


Using the 8XC751 microcontroller as an I²C bus master

AN422

```

0221 752204      434  SL2:      MOV      ByteCnt,#4      ;Set up byte count.
0224 11B9        435          ACALL   RcvSub
0226 2002F8      436          JB      Retry,SL2
                                437
0229 0529        438          INC     XmtDat
022B 052A        439          INC     XmtDat+1
022D 052B        440          INC     XmtDat+2
022F 052C        441          INC     XmtDat+3
0231 80CD        442          SJMP    MainLoop      ;Do it all again.
                                443
                                444          ENDASSEMBLY COMPLETE, 0 ERRORS FOUND

```

I2CAPP 83C751 Single Master I2C Routines

```

ACC. . . . . D ADDR 00E0H PREDEFINED
ATN. . . . . B ADDR 009EH PREDEFINED
BCARL. . . . . NUMB 0010H
BCDR. . . . . NUMB 0020H
BCSTP. . . . . NUMB 0004H
BCSTR. . . . . NUMB 0008H
BCXA. . . . . NUMB 0080H
BIDLE. . . . . NUMB 0040H NOT USED
BITCNT. . . . . D ADDR 0021H
BMRO. . . . . NUMB 0040H
BTIR. . . . . NUMB 0010H
BUSRESET. . . . . C ADDR 019BH
BXSTP. . . . . NUMB 0001H
BXSTR. . . . . NUMB 0002H
BYTECNT. . . . . D ADDR 0022H
CHEKLOOP. . . . . C ADDR 01B4H
CLRINT. . . . . C ADDR 0026H
CLRTI. . . . . B ADDR 00DDH PREDEFINED
CTVAL. . . . . NUMB 0002H
DRDY. . . . . B ADDR 009DH PREDEFINED
EA. . . . . B ADDR 00AFH PREDEFINED
ETI. . . . . B ADDR 00ABH PREDEFINED
FAULT. . . . . B ADDR 0001H
FIXBUS. . . . . C ADDR 01A4H
FIXBUSEX. . . . . C ADDR 01D7H
FLAGS. . . . . D ADDR 0020H
I2CFG. . . . . D ADDR 00D8H PREDEFINED
I2CON. . . . . D ADDR 0098H PREDEFINED
I2DAT. . . . . D ADDR 0099H PREDEFINED
MAINLOOP. . . . . C ADDR 0200H
MASTER. . . . . B ADDR 0099H PREDEFINED
MASTRQ. . . . . B ADDR 00DEH PREDEFINED
ML2. . . . . C ADDR 020BH
NOACK. . . . . B ADDR 0000H
P0. . . . . D ADDR 0080H PREDEFINED
RBIT. . . . . C ADDR 0153H
RCVBYTE. . . . . C ADDR 014FH
RCVDAT. . . . . D ADDR 0025H
RCVDATA. . . . . C ADDR 004EH
RCVSUB. . . . . C ADDR 00B9H
RDACK. . . . . C ADDR 0143H
RDAT. . . . . B ADDR 009FH PREDEFINED
RDATERR. . . . . C ADDR 0086H
RDERR. . . . . C ADDR 0163H
RDEX. . . . . C ADDR 0083H
RDLAST. . . . . C ADDR 0074H
RDLOOP. . . . . C ADDR 006AH
RECOVER. . . . . C ADDR 018AH
REPSTART. . . . . C ADDR 017AH
RESET. . . . . C ADDR 01E1H
RETRY. . . . . B ADDR 0002H
RSEX. . . . . C ADDR 0107H

```

Using the 8XC751 microcontroller as an I²C bus master

AN422

RSLAST	C ADDR	00F8H	MOV	434	0031 782504
RSLOOP	C ADDR	00EEH	ACALL	435	0031 7180
RSTOP	C ADDR	01C4H	JB	436	0031 800328
RSUBERR	C ADDR	010AH		437	
SAERR	C ADDR	011DH	INC	438	0031 80232
SCL	B ADDR	0080H	INC	439	0031 80232
SDA	B ADDR	0081H	INC	440	0031 80232
SDATERR	C ADDR	004CH	INC	441	0031 80232
SDELAY	C ADDR	01D8H	2INC	442	0031 8000
SDEX	C ADDR	0049H		443	
SDLOOP	C ADDR	003CH		444	
SENDAD2	C ADDR	0115H			
SENDADDR	C ADDR	010CH			
SENDATA	C ADDR	0027H			
SENDSTOP	C ADDR	0166H			
SENDSUB	C ADDR	0088H			
SL1	C ADDR	0213H			
SL2	C ADDR	0221H			
SLVADR	D ADDR	0023H			
SP	D ADDR	0081H	PREDEFINED		
SSEX	C ADDR	00B4H			
SSLOOP	C ADDR	00A7H			
SSUBERR	C ADDR	00B7H			
STACKSAVE	D ADDR	002DH			
SUBADR	D ADDR	0024H			
TIMER1	C ADDR	001BH	NOT USED		
TIRUN	B ADDR	00DCH	PREDEFINED		
XMBIT	C ADDR	0128H			
XMBIT2	C ADDR	012AH			
XMBX	C ADDR	013FH			
XMERR	C ADDR	0140H			
XMITADDR	C ADDR	0120H			
XMITBYTE	C ADDR	0125H			
XMTDAT	D ADDR	0029H			

Interfacing the PCD8584 I²C-bus controller to 80C51 family microcontrollers

AN425

DESCRIPTION

This application note shows how to use the PCD8584 I²C-bus controller with 80C51 family microcontrollers. One typical way of connecting the PCD8584 to an 80C31 is shown. Some basic software routines are described showing how to transmit and receive bytes in a single master system. An example is given of how to use these routines in an application that makes use of the I²C circuits on an I²C demonstration board.

The PCD8584 is used to interface between parallel microprocessor or microcontroller buses and the serial I²C bus. For a description of the I²C bus protocol refer to the I²C bus specification which is printed in the microcontroller user guide.

The PCD8584 controls the transmission and reception of data on the I²C bus, arbitration, clock speeds and transmission and reception of data on the parallel bus. The parallel bus is compatible with 80C51, 68000, 8085 and Z80 buses. Communication with the I²C-bus can be done on an interrupt or polled basis. This application note focuses on interfacing with 8051 microcontrollers in single master systems.

PCD8584

In Figure 1, a block diagram is shown of the PCD8584. Basically it consists of an I²C-interface similar to the one used in 84Cxx family microcontrollers, and a control block for interfacing to the microcontroller.

The control block can automatically determine whether the control signals are from 80xx or 68xxx type of microcontrollers.

This is determined after the first write action from the microcontroller to the PCD-8584. The control block also contains a programmable divider which allows the selection of different PCD8584 and I²C clocks.

The I²C interface contains several registers which can be written and read by the microcontroller.

S1 is the control/status register. This register is accessed while the A0 input is 1. The meaning of the bits depends on whether the register is written to or read from. When used

as a single master system the following bits are important:

PIN: Interrupt bit. This bit is made active when a byte is sent/received to/from the I²C-bus. When ENI is made active, PIN also controls the external INT line to interrupt the microcontroller.

ES0-ES2: These bits are used as pointer for addressing S0, S0', S2 and S3. Setting ES0 also enables the Serial I/O.

ENI: Enable Interrupt bit. Setting this bit enables the generation of interrupts on the INT line.

STA, STO: These bits allow the generation of START or STOP conditions.

ACK: With this bit set and the PCD8584 is in master/receiver mode, no acknowledge is generated by the PCD8584. The slave/transmitter now knows that no more data must be sent to the I²C-bus.

BER: This bit may be read to check if bus errors have occurred.

BB: This bit may be read to check whether the bus is free for I²C-bus transmission.

S2 is the clock register. It is addressed when A0 = 0 and ES0-ES2 = 010 in the previous write cycle to S1. With the bits S24-S20 it is possible to select 5 input clock frequencies and 4 I²C clock frequencies.

S3 is the interrupt vector register. It is addressed when A0 = 0 and ES0-ES2 = 001 in the previous write cycle to S1. This register is not used when an 80C51 family microcontroller is used. An 80C51 microcontroller has fixed interrupt vector addresses.

S0' is the own address register. It is addressed when A0 = 0 and ES0-ES2 = 000. This register contains the slave address of the PCD8584. In the single master system described here, this register has no functional use. However, by writing a value to S0', the PCD8584 determines whether an 80Cxx or 68xxx type microcontroller is the controlling microcontroller by looking at the CS and WR lines. So independent of whether the PCD8584 is used as master or slave, the

microcontroller should always first write a value to S0' after reset.

S0 is the I²C data register. It is addressed when A0 = 0 and ES0-ES2 = 1x0. Transmission of a byte on the I²C bus is done by writing this byte to S0. When the transmission is finished, the PIN bit in S1 is reset and if ENI is set, an interrupt will be generated. Reception of a byte is signaled by resetting PIN and by generating an interrupt if ENI is set. The received byte can be read from S0.

The SDA and SCL lines have no protection diodes to V_{DD}. This is important for multimaster systems. A system with a PCD8584 can now be switched off without causing the I²C-bus to hang-up. Other masters still can use the bus.

For more information of the PCD8584 refer to the data sheet.

PCD8584/8031 Hardware Interface

Figure 2 shows a minimum system with an 8051 family controller and a PCD8584. In this example, an 80C31 is used. However any 80C51 family controller with external addressing capability can be used.

The software resides in EPROM U3. For addressing this device, latch U2 is necessary to demultiplex the lower address bits from the data bits. The PCD8584 is mapped in the external data memory area. It is selected when A1 = 0. Because in this example no external RAM or other mapped peripherals are used, no extra address decoding components are necessary. A0 is used by the PCD8584 for proper register selection in the PCD8584.

U5A is an inverter with Schmitt trigger input and is used to buffer the oscillator signal of the microcontroller. Without buffering, the rise and fall time specifications of the CLK signal are not met. It is also important that the CLK signal has a duty cycle of 50%. If this is not possible with certain resonators or microcontrollers, then an extra flip-flop may be necessary to obtain the correct duty cycle.

U5C and U5D are used to generate the proper reset signals for the microcontroller and the PCD8584.

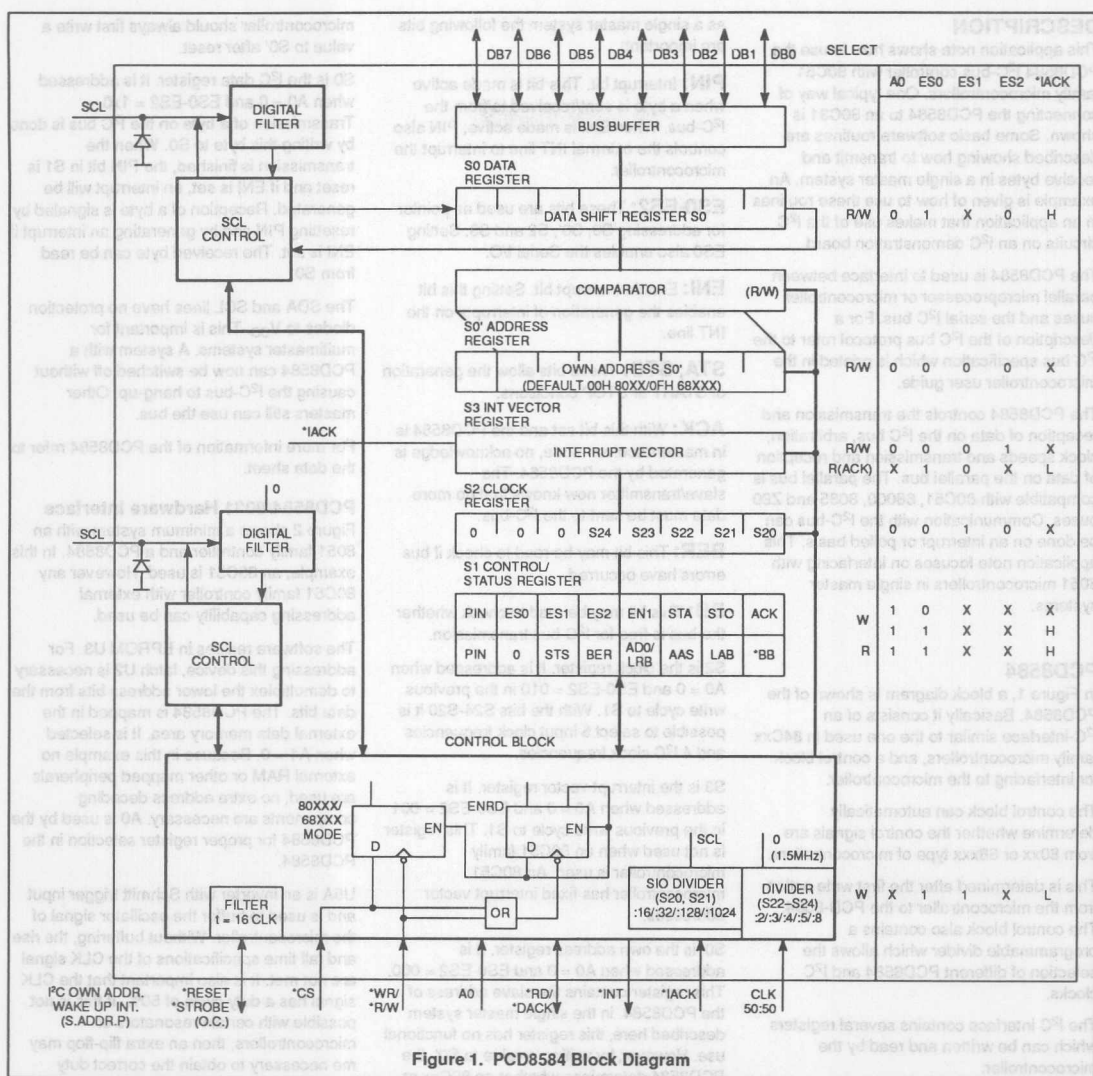


Figure 1. PCD8584 Block Diagram

Interfacing the PCD8584 I²C-bus controller to 80C51 family microcontrollers

AN425

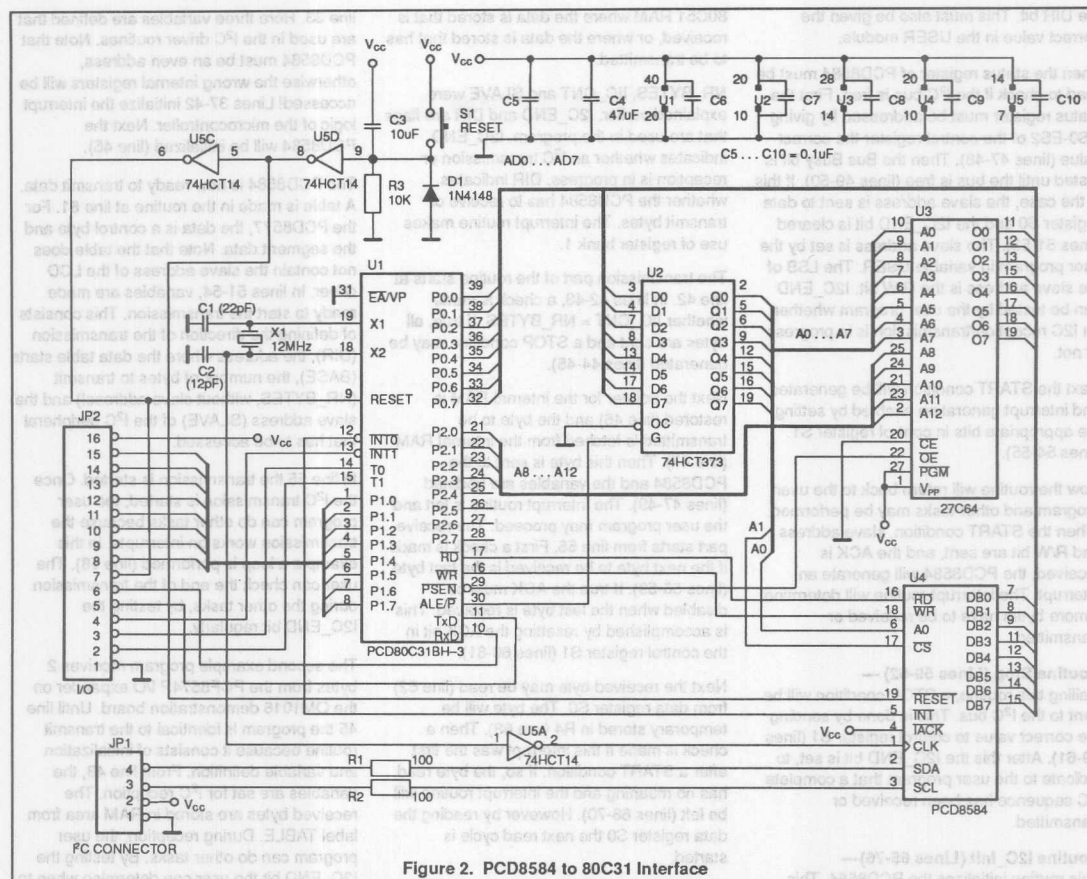


Figure 2. PCD8584 to 80C31 Interface

Basic PCD8584/8031 Driver Routines

In the listing section (page 1-25), some basic routines are shown. The routines are divided in two modules. The module ROUTINE contains the driver routines and initialization of the PCD8584. The module INTERR contains the interrupt handler. These modules may be linked to a module with the user program that uses the routines in INTERR and ROUTINE. In this application note, this module will be called USER. A description of ROUTINE and INTERR follows.

Module ROUTINE

Routine Sendbyte (Lines 17-20)

This routine sends the contents of the accumulator to the PCD8584. The address is such that A0 = 0. Which register is accessed depends on the contents of ES0-ES2 of the

control register. The address of the PCD8584 is in variable 'PCD8584'. This must have been previously defined in the user program. The DPTR is used as a pointer for addressing the peripheral. If the address is less than 255, then R0 or R1 may be used as the address pointer.

Routine Sendcontr (Lines 25, 26)

This routine is similar to Sendbyte, except that now A0 = 1. This means that the contents of the accumulator are sent to the control register S1 in the PCD8584.

Routine Readbyte (Lines 30-33)

This routine reads a register in the PCD8584, with A0 = 0. Which register depends on ES0-ES2 of the control register. The result of the read operation is returned in the accumulator.

Routine Readcontr (Lines 37-39)

This routine is similar to Readbyte, except that now A0 = 1. This means that the accumulator will contain the value of status register S1 of the PCD8584.

Routine Start Lines (44-56)

This routine generates a START-condition and the slave address with a R/W bit. In line 44, the variable IIC_CNT is reset. This variable is used as a byte counter to keep track of the number of bytes that are received or transmitted. IIC_CNT is defined in module INTERR.

Lines 45-46 increment the variable NR_BYTES if the PCD8584 must receive data. NR_BYTES is a variable that indicates how many bytes have to be received or transmitted. It must be given the correct value in the USER module. Receiving or transmitting is distinguished by the value of

Interfacing the PCD8584 I²C-bus controller to 80C51 family microcontrollers

AN425

the DIR bit. This must also be given the correct value in the USER module.

Then the status register of PCD8584 must be read to check if the I²C bus is free. First the status register must be addressed by giving ES0-ES2 of the control register the correct value (lines 47-48). Then the Bus Busy bit is tested until the bus is free (lines 49-50). If this is the case, the slave address is sent to data register S0 and the I2C_END bit is cleared (lines 51-53). The slave address is set by the user program in variable USER. The LSB of the slave address is the R/W bit. I2C_END can be tested by the user program whether an I2C reception/transmission is in progress or not.

Next the START condition will be generated and interrupt generation enabled by setting the appropriate bits in control register S1. (lines 54-55).

Now the routine will return back to the user program and other tasks may be performed. When the START condition, slave address and R/W bit are sent, and the ACK is received, the PCD8584 will generate an interrupt. The interrupt routine will determine if more bytes have to be received or transmitted.

Routine Stop (Lines 59-62) —

Calling this routine, a STOP condition will be sent to the I²C bus. This is done by sending the correct value to control register S1 (lines 59-61). After this the I2C_END bit is set, to indicate to the user program that a complete I²C sequence has been received or transmitted.

Routine I2C_Init (Lines 65-76)—

This routine initializes the PCD8584. This must be done directly after reset. Lines 67-70 write data to 'own address' register S0'. First the correct address of S0' is set in control register S1 (lines 67-68), then the correct value is written to it (lines 69-70). The value for S0' is in variable SLAVE_ADR and set by the user program. As noted previously, register S0' must always be the first register to be accessed after reset, because the PCD8584 now determines whether an 80Cxxx or 68xxx microcontroller is connected. Lines 72-76 set the clock register S2. The variable I2C_CLOCK is also set by the user program.

Module INTERRUPT

This module contains the I²C interrupt routine. This routine is called every time a byte is received or transmitted on the I²C bus. In lines 12-15 RAM space for variables is reserved.

BASE is the start address in the internal

80C51 RAM where the data is stored that is received, or where the data is stored that has to be transmitted.

NR_BYTES, IIC_CNT and SLAVE were explained earlier. I2C_END and DIR are flags that are used in the program. I2C_END indicates whether an I²C transmission or reception is in progress. DIR indicates whether the PCD8584 has to receive or transmit bytes. The interrupt routine makes use of register bank 1.

The transmission part of the routine starts at line 42. In lines 42-43, a check is made whether IIC_CNT = NR_BYTES. If true, all bytes are sent and a STOP condition may be generated (lines 44-45).

Next the pointer for the internal RAM is restored (line 46) and the byte to be transmitted is fetched from the internal RAM (line 47). Then this byte is sent to the PCD8584 and the variables are updated (lines 47-49). The interrupt routine is left and the user program may proceed. The receive part starts from line 55. First a check is made if the next byte to be received is the last byte (lines 56-59). If true the ACK must be disabled when the last byte is received. This is accomplished by resetting the ACK bit in the control register S1 (lines 60-61).

Next the received byte may be read (line 62) from data register S0. The byte will be temporary stored in R4 (line 63). Then a check is made if this interrupt was the first after a START condition. If so, the byte read has no meaning and the interrupt routine will be left (lines 68-70). However by reading the data register S0 the next read cycle is started.

If valid data is received, it will be stored in the internal RAM addressed by the value of BASE (lines 71-73). Finally a check is made if all bytes are received. If true, a STOP condition will be sent (lines 75-78).

EXAMPLES

In the listing section (starting on page 8), some examples are shown that make use of the routines described before. The examples are transmission of a sequence, reception of I²C data and an example that combines both.

The first example sends bytes to the PCD8577 LCD driver on the OM1016 demonstration board. Lines 7 to 10 define the interface with the other modules and should be included in every user program. Lines 14 to 16 define the segments in the user module. It is completely up to the user how to organize this.

Lines 24 and 28 are the reset and interrupt vectors. The actual use, program starts at

line 33. Here three variables are defined that are used in the I²C driver routines. Note that PCD8584 must be an even address, otherwise the wrong internal registers will be accessed! Lines 37-42 initialize the interrupt logic of the microcontroller. Next the PCD8584 will be initialized (line 45).

The PCD8584 is now ready to transmit data. A table is made in the routine at line 61. For the PCD8577, the data is a control byte and the segment data. Note that the table does not contain the slave address of the LCD driver. In lines 51-54, variables are made ready to start the transmission. This consists of defining the direction of the transmission (DIR), the address where the data table starts (BASE), the number of bytes to transmit (NR_BYTES, without slave address!) and the slave address (SLAVE) of the I²C peripheral that has to be accessed.

In line 55 the transmission is started. Once the I²C transmission is started, the user program can do other tasks because the transmission works on interrupts. In this example a loop is performed (line 58). The user can check the end of the transmission during the other tasks, by testing the I2C_END bit regularly.

The second example program receives 2 bytes from the PCF8574P I/O expander on the OM1016 demonstration board. Until line 45 the program is identical to the transmit routine because it consists of initialization and variable definition. From line 48, the variables are set for I²C reception. The received bytes are stored in RAM area from label TABLE. During reception, the user program can do other tasks. By testing the I2C_END bit the user can determine when to start processing the data in the TABLE.

The third example program displays time from the PCF8583P clock/calendar/RAM on the LCD display driven by the PCF8577. The LED display (driven by SAA1064) shows the value of the analog inputs of the A/D converter PCF8591. The four analog inputs are scanned consecutively.

In this example, both transmit and receive sequences are implemented as shown in the previous examples. The main clock part is from lines 62-128. This contains the calls to the I²C routines. From lines 135-160, routines are shown that prepare the data to be transmitted. Lines 171 to 232 are the main program for the AD converter and LED display. Lines 239 to 340 contain routines used by the main program. This demo program can also be used with the I²C peripherals on the OM1016 demonstration board.

Interfacing the PCD8584 I²C-bus controller to 80C51 family microcontrollers

AN425

ASM51 TSW ASSEMBLER Routines for PCD8584

```

LOC  OBJ      LINE  SOURCE
1  $TITLE (Routines for PCD8584)
2  $PAGELENGTH(40)
3  ;Program written for PCD8584 as master
4  ;
5      PUBLIC READBYTE,READCONTR,SENDBYTE
6      PUBLIC SENDCONTR,START,STOP
7      EXTRN BIT(I2C_END,DIR)
8      EXTRN DATA(SLAVE,IIC_CNT,NR_BYTES)
9      EXTRN NUMBER(SLAVE_ADR,I2C_CLOCK,PCD8584)
10 ;
11 ;Define code segment
12 ROUTINE  SEGMENT CODE
13         RSEG  ROUTINE
14 ;
15 ;SENDBYTE sends a byte to PCD8584 with A0=0
16 ;Byte to be send must be in accu
0000:      R 17 SENDBYTE:
0000: 900000  R 18     MOV DPTR,#PCD8584 ;Register address
0003: F0      19 SEND:  MOVX @DPTR,A    ;Send byte
0004: 22      20     RET
21 ;
22 ;SENDCONTR sends a byte to PCD8584 with A0=1
23 ;Byte to be send must be in accu
0005:      24 SENDCONTR:
0005: 900001  R 25     MOV DPTR,#PCD8584+01H ;Register address
0008: 80F9      26     JMP SEND
27 ;
28 ;READBYTE reads a byte from PCD8584 with A0=0
29 ;Received byte is stored in accu
000A:      30 READBYTE:
000A: 900000  R 31     MOV DPTR,#PCD8584 ;Register address
000D: E0      32 REC:  MOVX A,@DPTR    ;Receive byte
000E: 22      33     RET
34 ;
35 ;READCONTR reads a byte from PCD8584 with A0=1
36 ;Received byte is stored in accu
000F:      37 READCONTR:
000F: 900001  R 38     MOV DPTR,#PCD8584+01H ;Register address
0012: 80F9      39     JMP REC
40 ;
41 ;START tests if the I2C bus is ready. If ready a
42 ;START-condition will be sent, interrupt generation
43 ;and acknowledge will be enabled.
0014: 750000  R 44 START: MOV IIC_CNT,#00 ;Clear I2C byte counter
0017: 200002  R 45     JB DIR,PROCEED ;If DIR is 'receive' then
001A: 0500      46     INC NR_BYTES ;increment NR_BYTES
001C: 7440      47 PROCEED:MOV A,#40H ; Read STATUS register of
; 8584
001E: 120005  R 48     CALL SENDCONTR
0021: 12000F  R 49 TESTBB: CALL READCONTR
0024: 30E0FA  50     JNB ACC.0,TESTBB; Test BB/ bit
0027: E500      51     MOV A,SLAVE
0029: C200      52     CLR I2C_END ;Reset I2C ready bit
002B: 120000  R 53     CALL SENDBYTE ;Send slave address
002E: 744D      54     MOV A,#01001101B;Generate START, set ENI,
;set ACK

```

Interfacing the PCD8584 I²C-bus controller to 80C51 family microcontrollers

AN425

0030:	120005	R	55	CALL SENDCONTR
0033:	22		56	RET
			57	;
			58	;STOP will generate a STOP condition and set the ;I2C_END bit
0034:	74C3		59	STOP: MOV A,#11000011B
0036:	120005	R	60	CALL SENDCONTR ;Send STOP condition
0039:	D200	R	61	SETB I2C_END ;Set I2C_END bit
003B:	22		62	RET
			63	;
			64	;I2C_init does the initialization of the PCD8584
003C:			65	I2C_INIT:
			66	;Write own slave address
003C:	E4		67	CLR A
003D:	120005	R	68	CALL SENDCONTR ;Write to control register
0040:	7400	R	69	MOV A,#SLAVE_ADR
0042:	120000	R	70	CALL SENDBYTE ;Write to own slave ;register
			71	;Write clock register
0045:	7420		72	MOV A,#20H
0047:	120005	R	73	CALL SENDCONTR ;Write to control register
004A:	7400	R	74	MOV A,#I2C_CLOCK
004C:	120000	R	75	CALL SENDBYTE ;Write to clock register
004F:	22		76	RET
			77	;
0050:			78	END

Interfacing the PCD8584 I²C-bus controller to 80C51 family microcontrollers

AN425

LOC	OBJ	LINE	SOURCE
ASMS1	TSW	ASSEMBLER	I2C INTERRUPT ROUTINE
		1	\$TITLE (I2C INTERRUPT ROUTINE)
		2	\$PAGELENGTH(40)
		3	;
		4	PUBLIC INTO_SRV
		5	PUBLIC DIR,I2C_END
		6	PUBLIC BASE,NR_BYTES,IIC_CNT,SLAVE
		7	EXTRN CODE (SENDBYTE,SENDCONTR,STOP)
		8	EXTRN CODE (READBYTE,READCONTR)
		9	;
		10	;Define variables in RAM
		11	IIC_VAR SEGMENT DATA
		12	RSEG IIC_VAR
0000:	R	12	BASE: DS 1 ;Pointer to I2C table (till
		13	NR_BYTES: DS 1 ;Number of bytes to rcv/trm
0001:		14	IIC_CNT:DS 1 ;I2C byte counter
0002:		15	SLAVE: DS 1 ;Slave address after START
0003:		16	;
		17	;Define variable segment
		18	BIT_VAR SEGMENT DATA BITADDRESSABLE
		19	RSEG BIT_VAR
0000:	R	20	STATUS: DS 1 ;Byte with flags
0000	R	21	I2C_END BIT STATUS.0 ;Defines if a I2C
		22	;transmission is finished
		23	; '1' is finished
		24	; '0' is not ready
0000	R	24	DIR BIT STATUS.3 ;Defines direction of I2C
		25	;transmission
		26	; '1':Transmit '0':Receive
		27	;
		28	;Define code segment for routine
		29	IIC_INT SEGMENT CODE PAGE
		30	RSEG IIC_INT
		31	;
		32	;Program uses registers in RB1
		33	USING 1
		34	;
0000:	R	34	INTO_SRV:
0000: C0E0		35	PUSH ACC ;Save acc. en psw on stack
0002: C0D0		36	PUSH PSW
0004: 75D008		37	MOV PSW,#08H ;Select register bank 1
0007: 300016	R	38	JNB DIR,RECEIVE ;Test direction bit
		39	;8584 is MST/TRM
		40	;
		41	;Program part to transmit bytes to IIC bus
000A: E502	R	42	MOV A,IIC_CNT ;Compare IIC_CNT and
		43	NR_BYTES
000C: B50105	R	43	CJNE A,NR_BYTES,PROCEED
000F: 120000	R	44	CALL STOP ;All bytes transmitted
0012: 8032		45	JMP EXIT
0014: A800	R	46	PROCEED:MOV R0,BASE ;RAM pointer
0016: E6		47	MOV A,@R0 ;Source is internal RAM
0017: 0500	R	48	INC BASE ;Update pointer of table
0019: 120000	R	49	CALL SENDBYTE ;Send byte to IIC bus
001C: 0502	R	50	INC IIC_CNT ;Update byte counter
001E: 8026		51	JMP EXIT

Interfacing the PCD8584 I²C-bus controller to 80C51 family microcontrollers

AN425

Address	Disassembly	Comments
0020:	52 ;	
0020: E502	53 ;	
0022: 04	54 ;Program to receive byte from IIC bus	
0023: 04	55 RECEIVE:	
0024: B50105	56 MOV A,IIC_CNT ;Test if last byte is to be	
0027: 7448	57 INC A ;received	
0029: 120000	58 INC A	
002C: 120000	59 CJNE A,NR_BYTES,PROC_RD	
002F: FC	60 MOV A,#01001000B;Last byte to be received.	
	61 CALL SENDCONTR;Write control word to	
	62 PROC_RD:CALL READBYTE ;Read I2C byte	
	63 MOV R4,A ;Save accu	
	64 ;If RECEIVE is entered after the transmission of	
	65 ;START+address then the result of READBYTE is not	
	66 ;relevant. READBYTE is used to start the generation	
	67 ;of the clock pulses for the next byte to read.	
0030: E4	68 ;This situation occurs when IIC_CNT is 0	
0031: B50202	69 CLR A ;Test IIC_CNT	
0034: 8006	70 CJNE A,IIC_CNT,SAVE	
	71 JMP END_TEST ;START is send. No relevant	
0036: A800	72 MOV R0,BASE ;data in data reg. of 8584	
0038: EC	73 SAVE: MOV R0,BASE	
0039: F6	74 MOV A,R4 ;Destination is internal RAM	
003A: 0500	75 MOV @R0,A	
003C: 0502	76 INC BASE	
	77 END_TEST:INC IIC_CNT;Test if all bytes are	
	78 ;received.	
003E: E501	79 MOV A,NR_BYTES	
0040: B50203	80 CJNE A,IIC_CNT,EXIT	
0043: 120000	81 CALL STOP ;All bytes received	
	82 ;	
0046: D0D0	83 EXIT: POP PSW ;Restore PSW and accu	
0048: D0D0	84 POP ACC	
004A: 32	85 RETI	
	86 ;	
004B:	87 END	

Interfacing the PCD8584 I²C-bus controller to 80C51 family microcontrollers

AN425

ASM51	TSW	ASSEMBLER	Send a string of bytes to the PCF8577 on OM1016	11	R	0000	:100
LOC	OBJ	LINE	SOURCE	22	R	0000	:100
		1	\$TITLE (Send a string of bytes to the PCF8577 on OM1016)	24	R	0000	:100
		2	\$PAGELENGTH(40)	25	R	0000	:100
		3	;	26	R	0000	:100
		4	;This program is an example to transmit bytes via PCD8584	27	R	0000	:100
		5	;to the I2C-bus	28	R	0000	:100
		6	;	29	R	0000	:100
		7	PUBLIC SLAVE_ADR,I2C_CLOCK,PCD8584	30	R	0000	:100
		8	EXTRN CODE(I2C_INIT,INT0_SRV,START)	31	R	0000	:100
		9	EXTRN BIT(I2C_END,DIR)	32	R	0000	:100
		10	EXTRN DATA(BASE,NR_BYTES,IIC_CNT,SLAVE)	33	R	0000	:100
		11	;	34	R	0000	:100
		12	;	35	R	0000	:100
		13	;Define used segments	36	R	0000	:100
		14	USER SEGMENT CODE ;Segment for user program	37	R	0000	:100
		15	RAMTAB SEGMENT DATA ;Segment for table in	38	R	0000	:100
			;internal RAM	39	R	0000	:100
		16	RAMVAR SEGMENT DATA ;Segment for RAM variables	40	R	0000	:100
			;in RAM	41	R	0000	:100
		17	;	42	R	0000	:100
		18	;	43	R	0000	:100
		19	RSEG RAMVAR	44	R	0000	:100
0000:	R	20	STACK: DS 20 ;Reserve stack area	45	R	0000	:100
		21	;	46	R	0000	:100
		22	;	47	R	0000	:100
		23	CSEG AT 00H	48	R	0000	:100
0000: 020000	R	24	JMP MAIN ;Reset vector	49	R	0000	:100
		25	;	50	R	0000	:100
		26	;	51	R	0000	:100
		27	CSEG AT 03H	52	R	0000	:100
0003: 020000	R	28	JMP INTO_SRV ;I2C interrupt vector	53	R	0000	:100
			;(INT0/)	54	R	0000	:100
		29	;	55	R	0000	:100
		30	;	56	R	0000	:100
		31	RSEG USER	57	R	0000	:100
		32	;Define I2C clock, own slave address and PCD8584	58	R	0000	:100
			;hardware address	59	R	0000	:100
0055		33	SLAVE_ADR EQU 55H ;Own slave address is 55H	60	R	0000	:100
001C		34	I2C_CLOCK EQU 00011100B ;12.00MHz/90kHz	61	R	0000	:100
0000		35	PCD8584 EQU 0000H ;PCD8584 address with A0=0	62	R	0000	:100
		36	;0000: 7581FF R 37 MAIN: MOV SP,#STACK-1 ;Initialize stack pointer	63	R	0000	:100
		38	;Initialize 8031 interrupt registers for I2C	64	R	0000	:100
			;interrupt	65	R	0000	:100
0003: D2A8		39	SETB EX0 ;Enable interrupt INT0/	66	R	0000	:100
0005: D2AF		40	SETB EA ;Set global enable	67	R	0000	:100
0007: D2B8		41	SETB PX0 ;Priority level '1'	68	R	0000	:100
0009: D288		42	SETB IT0 ;INT0/ on falling edge	69	R	0000	:100
		43	;	70	R	0000	:100
		44	;Initialize PCD8584	71	R	0000	:100
000B: 120000	R	45	CALL I2C_INIT	72	R	0000	:100
		46	;	73	R	0000	:100
		47	;Make a table in RAM with data to be transmitted.	74	R	0000	:100
000E: 120021	R	48	CALL MAKE_TAB	75	R	0000	:100
		49	;	76	R	0000	:100
		50	;Set variables to control PCD8584	77	R	0000	:100

Interfacing the PCD8584 I²C-bus controller to 80C51 family microcontrollers

AN425

0011: D200	R	51	SETB DIR	;DIR='transmission'
0013: 750000	R	52	MOV BASE,#TABLE	;Start address of I2C-data
0016: 750005	R	53	MOV NR_BYTES,#05H	;5 bytes must be transferred
0019: 750074	R	54	MOV SLAVE,#01110100B	;Slave address PCF8577
				; + WR/
001C: 120000	R	55	CALL START	;Start I2C transmission
		56	;	
		57	;	
001F: 80FE		58	LOOP: JMP LOOP	;Endless loop when program is finished
		59	;	
		60	;	
0021:		61	MAKE_TAB:	
0021: 7800	R	62	MOV R0,#TABLE	;Make data ready for I2C
				;transmission
0023: 7600		63	MOV @R0,#00	;Controlword PCF8577
0025: 08		64	INC R0	
0026: 76FC		65	MOV @R0,#0FCH	; '0'
0028: 08		66	INC R0	
0029: 7660		67	MOV @R0,#60H	; '1'
002B: 08		68	INC R0	
002C: 76DA		69	MOV @R0,#0DAH	; '2'
002E: 08		70	INC R0	
002F: 76F2		71	MOV @R0,#0F2H	; '3'
0031: 22		72	RET	
		73	;	
		74	;	
----		75	RSEG RAMTAB	
0000:	R	76	TABLE: DS 10	;Reserve space in internal
				;data RAM
		77		;for I2C data to transmit
		78	;	
		79	;	
000A:		80	END	

Interfacing the PCD8584 I²C-bus controller to 80C51 family microcontrollers

AN425

```

ASM51  TSW  ASSEMBLER  Receive 2 bytes from the PCF8574P on OM1016:  52  R  0000: 120000
LOC  OBJ  LINE  SOURCE
1  $TITLE (Receive 2 bytes from the PCF8574P on OM1016)
2  $PAGELENGTH(40)
3  ;
4  ;This program is an example to receive bytes via
   PCD8584
5  ;from the I2C-bus
6  ;
7      PUBLIC  SLAVE_ADR,I2C_CLOCK,PCD8584
8      EXTRN  CODE(I2C_INIT,INT0_SRV,START)
9      EXTRN  BIT(I2C_END,DIR)
10     EXTRN  DATA(BASE,NR_BYTES,IIC_CNT,SLAVE)
11 ;
12 ;
13 ;Define used segments
14 USER  SEGMENT CODE      ;Segment for user program
15 RAMTAB SEGMENT DATA    ;Segment for table in
                           ;internal RAM
16 RAMVAR SEGMENT DATA    ;Segment for RAM variables
                           ;in RAM
17 ;
18 ;
----
19      RSEG  RAMVAR
0000:  R  20  STACK:  DS 20      ;Reserve stack area
21 ;
22 ;
----
23      CSEG AT 00H
0000: 020000  R  24      JMP MAIN      ;Reset vector
25 ;
26 ;
----
27      CSEG AT 03H
0003: 020000  R  28      JMP INT0_SRV    ;I2C interrupt vector
                           ;(INT0/)
29 ;
30 ;
----
31      RSEG USER
32 ;Define I2C clock, own slave address and PCD8584
   ;hardware address
0055  33  SLAVE_ADR EQU 55H      ;Own slave address is 55H
001C  34  I2C_CLOCK EQU 00011100B ;12.00MHz/90kHz
0000  35  PCD8584 EQU 0000H      ;PCD8584 address with A0=0
36 ;0000: 7581FF  R  37  MAIN:  MOV SP,#STACK-1 ;Initialize stack pointer
38 ;Initialize 8031 interrupt registers for I2C
   ;interrupt
0003: D2A8  39      SETB EX0      ;Enable interrupt INT0/
0005: D2AF  40      SETB EA      ;Set global enable
0007: D2B8  41      SETB PX0      ;Priority level '1'
0009: D288  42      SETB IT0      ;INT0/ on falling edge
43 ;
44 ;Initialize PCD8584
000B: 120000  R  45      CALL I2C_INIT
46 ;
47 ;Set variables to control PCD8584
000E: C200  R  48      CLR DIR      ;DIR='receive'
0010: 750000  R  49      MOV BASE,#TABLE ;Start address of I2C-data
0013: 750002  R  50      MOV NR_BYTES,#02H ;2 bytes must be received
0016: 75004F  R  51      MOV SLAVE,#01001111B ;Slave address PCF8574
                           ; + RD

```

Interfacing the PCD8584 I²C-bus controller to 80C51 family microcontrollers

AN425

```

0019: 120000  R  52      CALL START      ;Start I2C transmission
                    53 ;
                    54 ;
001C: 80FE      55  LOOP:  JMP LOOP      ;Endless loop when program
                    ;is finished
                    56 ;
                    57 ;
----          58      RSEG RAMTAB      ;This program is an example to receive I2C data
0000:          R  59  TABLE: DS 10      ;Reserve space in internal
                    ;data RAM
                    60      ;for received I2C data
                    61 ;
                    62 ;
000A:          63      END
                    EXTRN DATA_HASE,IN_BYTES,IIC_CNT,SLAVE
                    EXTRN BIT(I2C_END,DIR)
                    EXTRN CODE(I2C_WAIT,INTO_SRV,START)
                    PUBLIC SLAVE_ADDR,I2C_CLOCK,PCD8584
                    1  STILL (receive I by I2C)
                    2  LABEL:ENDTH(40)
                    3  ;
                    4  ;This program is an example to receive I2C data
                    5  ;from the I2C-bus
                    6  ;
                    7  ;
                    8  ;
                    9  ;
                   10  EXTRN DATA_HASE,IN_BYTES,IIC_CNT,SLAVE
                   11  ;
                   12  ;
                   13  ;Define used segments
                   14  USER  SEGMENT CODE      ;Segment for user program
                   15  RAMTAB SEGMENT DATA      ;Segment for table in
                   16  RAMVAR  SEGMENT DATA      ;Segment for RAM variables
                   17  ;
                   18  ;
                   19  ;
                   20  STACK: DS 20      ;Reserve stack area
                   21  ;
                   22  ;
                   23  CSEG AT 00H
                   24  LBR MAIN
                   25  ;
                   26  ;
                   27  CSEG AT 01H
                   28  JMP INTO_SRV      ;I2C interrupt vector
                   29  ;
                   30  ;
                   31  ;
                   32  RSEG USER
                   33  ;Define I2C clock, own slave address and PCD8584
                   34  ;hardware address
                   35  SLAVE_ADDR EQU 00H 52H
                   36  I2C_CLOCK EQU 0011000H ;12.00MHz/100kHz
                   37  PCD8584 EQU 0000H ;PCD8584 address with A0=0
                   38  ;
                   39  ;
                   40  ;
                   41  ;
                   42  ;
                   43  ;
                   44  ;Initialize PCD8584
                   45  CALL I2C_INIT
                   46  ;
                   47  ;Set variables to control PCD8584
                   48  CLR DIR
                   49  ;DIR=receive
                   50  MOV BASE,TABLE ;Start address of I2C-data
                   51  MOV IN_BYTES,02H ;2 bytes must be received
                   52  MOV SLAVE,#01001111B ;Slave address PCD8584
                   53  ;
                   54  ;
                   55  ;
                   56  ;
                   57  ;
                   58  ;
                   59  ;
                   60  ;
                   61  ;
                   62  ;
                   63  ;
                   64  ;
                   65  ;
                   66  ;
                   67  ;
                   68  ;
                   69  ;
                   70  ;
                   71  ;
                   72  ;
                   73  ;
                   74  ;
                   75  ;
                   76  ;
                   77  ;
                   78  ;
                   79  ;
                   80  ;
                   81  ;
                   82  ;
                   83  ;
                   84  ;
                   85  ;
                   86  ;
                   87  ;
                   88  ;
                   89  ;
                   90  ;
                   91  ;
                   92  ;
                   93  ;
                   94  ;
                   95  ;
                   96  ;
                   97  ;
                   98  ;
                   99  ;
                  100  ;

```

Interfacing the PCD8584 I²C-bus controller to 80C51 family microcontrollers

AN425

LOC	OBJ	LINE	SOURCE	TIME	CHG	LOC
ASM51	TSW	ASSEMBLER	Demo program for PCD8584 I2C-routines			ASM51
		1	\$TITLE (Demo program for PCD8584 I2C-routines)	36		0000
		2	\$PAGELENGTH(40)			
		3	;Program displays on the LCD display the time (with	14		0078
			;PCF8583). Dots on LCD display blink every second.			
		5	;On the LED display the values of the successive	48		
			;analog input channels are shown.	48		0000
		7	;Program reads analog channels of PCF8591P.	50		
		8	;Channel number and channel value are displayed			
			;successively.	51		0003
		9	;Values are displayed on LCD and LED display on I2C	52		0005
			;demo board.	53		0007
		10	;	54		0009
		11	PUBLIC SLAVE_ADR,I2C_CLOCK,PCD8584	55		
		12	EXTRN CODE(I2C_INIT,INTO_SRV,START)	56		0000
		13	EXTRN BIT(I2C_END,DIR)	57		
		14	EXTRN DATA(BASE,NR_BYTES,IIC_CNT,SLAVE)	58		0000
		15	;	59		
		16	;	60		
		17	;Define used segments	61		
		18	USER SEGMENT CODE ;Segment for user program	62		
		19	RAMTAB SEGMENT DATA ;Segment for table in	63		0011
			;internal RAM	64		0013
		20	RAMVAR SEGMENT DATA ;Segment for variables	65		0015
		21	;	66		0017
		22	RSEG RAMVAR	67		
0000:	R	23	STACK: DS 20 ;Stack area (20 bytes)	68		0019
0014:		24	PREVIOUS: DS 1 ;Store for previous seconds	69		
0015:		25	CHANNEL:DS 1 ;Channel number to be	70		0021
			;sampled	71		
0016:		26	AN_VAL: DS 1 ;Analog value sampled	72		0023
			;channel	73		0025
0017:		27	CONVAL: DS 3 ;Converted BCD value sampled	74		
			;channel	75		
		28	;	76		0027
----		29	CSEG AT 00H	77		0029
0000: 020000	R	30	LJMP MAIN ;Reset vector	78		0031
		31	;	79		0033
----		32	CSEG AT 03H ;INTO/	80		0035
0003: 020000	R	33	LJMP INTO_SRV ;Vector I2C-interrupt	81		0037
		34	;	82		0039
		35	;	83		0041
----		36	RSEG USER37 ;Define I2C clock, own slave address and address for	84		0043
			;main processor	85		
0055		38	SLAVE_ADR EQU 55H ;Own slaveaddress is 55h	86		0045
001C		39	I2C_CLOCK EQU 00011100B ;12.00MHz/90kHz	87		0047
0000		40	PCD8584 EQU 0000H ;Address of PCD8584. This	88		0049
			;must be an EVEN number!!	89		0051
00A3		41	;Define addresses of I2C peripherals	90		0053
		42	PCF8583R EQU 10100011B ;Address PCF8583 with Read	91		0055
			;active	92		0057
00A2		43	PCF8583W EQU 10100010B ;Address PCF8583 with Write	93		0059
			;active	94		0061
009F		44	PCF8591R EQU 10011111B ;Address PCF8591 with Read	95		0063
			;active	96		0065
009E		45	PCF8591W EQU 10011110B ;Address PCF8591 with Write	97		0067
			;active	98		0069


```

ASM51  TSW  ASSEMBLER  Demo program for PCD8584 I2C-routines
LOC  OBJ          LINE  SOURCE
0074          46  PCF8577W EQU 01110100B ;Address PCF8577 with Write
                                ;active
0076          47  SAA1064W EQU 01110110B ;Address SAA1064 with Write
                                ;active
                                ;On the I2C display the values of the successive
                                ;bytes on I2C display every second.
0000: 7581FF  R  49  MAIN:  MOV SP,#STACK-1 ;Define stack pointer
0003: D2A8    50  ;Initialize 80C31 interrupt registers for I2C
0005: D2AF    51  ;interrupt (INT0/)
0007: D2B8    52  SETB EX0      ;Enable interrupt INT0/
0009: D288    53  SETB EA      ;Set global enable
                                ;Values are also
000B: 120000  R  54  SETB PX0      ;Priority level is '1'
000E: 751500  R  55  SETB IT0      ;INT0/ on falling edge
                                ;Initialize PCD8584
000B: 120000  R  56  CALL I2C_INIT ;I2C INIT
000E: 751500  R  57  ;
                                ;MOV CHANNEL,#00 ;Set AD-channel
0011: D200    R  58  ;
0013: 750000  R  59  ;Time must be read from PCD8583.
0016: 750002  R  60  ;First write word address and control register of
0019: 7500A2  R  61  ;PCD8583.
001C: E4      62  SETB DIR      ;DIR='transmission'
001D: F500    R  63  MOV BASE,#TABLE ;Start address I2C data
001F: F501    R  64  MOV NR_BYTES,#02H ;Send 2 bytes
0021: 120000  R  65  MOV SLAVE,#PCF8583W
0024: 3000FD  R  66  CLR A
0027: D200    R  67  MOV TABLE,A ;Data to be sent (word 00
0029: 750000  R  68  ;address).
002C: 7500A2  R  69  MOV TABLE+1,A ;control
002F: 7401    70  ;byte)
0031: F500    R  71  CALL START ;Start transmission.
0033: F500    R  72  FIN_1: JNB I2C_END,FIN_1 ;Wait till transmission
0035: 120000  R  73  ;finished
0038: 3000FD  R  74  ;Send word address before reading time
003B: C200    R  75  REPEAT: SETB DIR ;'transmission'
003D: 750000  R  76  MOV BASE,#TABLE ;I2C data
0040: 7500A2  R  77  MOV SLAVE,#PCF8583W
0043: 7500A3  R  78  MOV A,#01
0046: 120000  R  79  MOV NR_BYTES,A ;Send 1 byte
0049: 3000FD  R  80  MOV TABLE,A ;Data to be sent is '1'
004C: 7800    R  81  CALL START ;Start I2C transmission
004E: 7902    R  82  FIN_2: JNB I2C_END,FIN_2 ;Wait till transmission
0050: E6      83  ;finished
0050: E6      84  ;
0050: E6      85  ;Time can now be read from PCD8583. Data read is
0050: E6      86  ;hundredths of sec's, sec's, min's and hr's
0050: E6      87  CLR DIR ;DIR='receive'
0050: E6      88  MOV BASE,#TABLE ;I2C table
0050: E6      89  MOV NR_BYTES,#04 ;4 bytes to receive
0050: E6      90  MOV SLAVE,#PCF8583R
0050: E6      91  CALL START ;Start I2C reception
0050: E6      92  FIN_3: JNB I2C_END,FIN_3 ;Wait till finished
0050: E6      93  ;
0050: E6      94  ;Transfer data to R2...R5
0050: E6      95  MOV R0,#TABLE ;Set pointers
0050: E6      96  MOV R1,#02H ;Pointer R2
0050: E6      97  TRANSFER:MOV A,@R0

```

Interfacing the PCD8584 I²C-bus controller to 80C51 family microcontrollers

AN425

LOC	OBJ	LINE	SOURCE	LOC	OBJ	LINE	SOURCE
0051: F7		94	MOV @R1,A	0091: EC		144	MOV A,R1
0052: 08		95	INC R0	0092: 13098	R	145	CALL LCD_DATA
0053: 09		96	INC R1	0093: 52		146	RET
0054: D500F9	R	97	DJNZ NR_BYTES,TRANSFER			147	
0057: ED		98	MOV A,R5			148	MOV A,R5
0058: 543F		99	ANL A,#3FH			149	ANL A,#3FH
005A: FD		100	MOV R5,ADDEND			150	MOV R5,ADDEND
		101	;			151	;
		102	;Data must now be displayed on LCD display.			152	;Data must now be displayed on LCD display.
		103	;First minutes and hours (in R4 and R5) must be			153	;First minutes and hours (in R4 and R5) must be
		104	;converted from BCD to LCD segment data.The segment			154	;converted from BCD to LCD segment data.The segment
			;data			155	;data
		105	;will be transferred to TABLE. R0 is pointer to			156	;will be transferred to TABLE. R0 is pointer to
			;table			157	;table
005B: 7800	R	106	MOV R0,#TABLE			158	MOV R0,#TABLE
005D: 7600		107	MOV @R0,#00H			159	MOV @R0,#00H
005F: 08		108	INC R0 0060: 120080			160	INC R0 0060: 120080
		109	CALL CONV			161	CALL CONV
		110	;			162	;
		111	;Switch on dp between hours and minutes			163	;Switch on dp between hours and minutes
0063: 430301	R	112	ORL TABLE+3,#01H			164	ORL TABLE+3,#01H
		113	;If lsb of seconds is '0' then switch on dp.			165	;If lsb of seconds is '0' then switch on dp.
0066: EB		114	MOV A,R3			166	MOV A,R3
0067: 13		115	RRC A			167	RRC A
0068: 4003		116	JC PROCEED			168	JC PROCEED
006A: 430101	R	117	ORL TABLE+1,#01H;switch on dp			169	ORL TABLE+1,#01H;switch on dp
		118	;			170	;
		119	;Now the time (hours,minutes) can be displayed on			171	;Now the time (hours,minutes) can be displayed on
			;the LCD			172	;the LCD
006D:		120	PROCEED:			173	PROCEED:
006D: D200	R	121	SETB DIR			174	SETB DIR
006F: 750000	R	122	MOV BASE,#TABLE			175	MOV BASE,#TABLE
0072: 750005	R	123	MOV NR_BYTES,#05H			176	MOV NR_BYTES,#05H
0075: 750074	R	124	MOV SLAVE,#PCF8577W			177	MOV SLAVE,#PCF8577W
0078: 120000	R	125	CALL START			178	CALL START
		126	;			179	;
007B: 3000FD	R	127	FIN_4: JNB I2C_END,FIN_4			180	FIN_4: JNB I2C_END,FIN_4
007E: 8026		128	JMP ADCON			181	JMP ADCON
			;Proceed with AD-conversion			182	;Proceed with AD-conversion
			;part			183	;part
		129	;			184	;
		130	*****			185	*****
		131	;Routines used by clock part of demo			186	;Routines used by clock part of demo
		132	;			187	;
		133	;CONV converts hour and minute data to LCD data and			188	;CONV converts hour and minute data to LCD data and
			;stores			189	;stores
		134	;it in TABLE.			190	;it in TABLE.
0080: 90009C	R	135	CONV: MOV DPTR,#LCD_TAB			191	CONV: MOV DPTR,#LCD_TAB
			;Base for LCD segment			192	;Base for LCD segment
			;table			193	;table
0083: ED		136	MOV A,R5			194	MOV A,R5
0084: C4		137	SWAP A			195	SWAP A
0085: 120096	R	138	CALL LCD_DATA			196	CALL LCD_DATA
			;Convert 10's hours to LCD			197	;Convert 10's hours to LCD
			;data in table			198	;data in table
0088: ED		139	MOV A,R5			199	MOV A,R5
0089: 120096	R	140	CALL LCD_DATA			200	CALL LCD_DATA
008C: EC		141	MOV A,R4			201	MOV A,R4
008D: C4		142	SWAP A			202	SWAP A
008E: 120096	R	143	CALL LCD_DATA			203	CALL LCD_DATA
			;Convert 10's minutes			204	;Convert 10's minutes

Interfacing the PCD8584 I²C-bus controller to 80C51 family microcontrollers

AN425

LOC	OBJ	LINE	SOURCE	LOC	OBJ	LINE	SOURCE
0091: EC		144	MOV A,R4	0092: 120096	R	145	CALL LCD_DATA ;Convert minutes
0095: 22		146	RET	0095: 22		146	RET
		147	;			147	;
		148	;LCD_DATA gets data from segment table and stores it			148	;LCD_DATA gets data from segment table and stores it
			;in TABLE				;in TABLE
0096: 540F		149	LCD_DATA:ANL A,#0FH ;Mask off LS-nibble	0096: 540F		149	LCD_DATA:ANL A,#0FH ;Mask off LS-nibble
0098: 93		150	MOVC A,@A+DPTR ;Get LCD segment data	0098: 93		150	MOVC A,@A+DPTR ;Get LCD segment data
0099: F6		151	MOV @R0,A ;Save data in table	0099: F6		151	MOV @R0,A ;Save data in table
009A: 08		152	INC R0	009A: 08		152	INC R0
009B: 22		153	RET	009B: 22		153	RET
		154	;			154	;
009C:		155	;LCD_TAB is conversion table for LCD	009C:		155	;LCD_TAB is conversion table for LCD
009C: FC60DA		156	LCD_TAB:	009C: FC60DA		156	LCD_TAB:
009F: F266B6		157	DB 0FCH,60H,0DAH; '0','1','2'	009F: F266B6		157	DB 0FCH,60H,0DAH; '0','1','2'
00A2: 3EE0FE		158	DB 0F2H,66H,0B6H; '3','4','5'	00A2: 3EE0FE		158	DB 0F2H,66H,0B6H; '3','4','5'
00A5: E6		159	DB 3EH,0E0H,0FEH; '6','7','8'	00A5: E6		159	DB 3EH,0E0H,0FEH; '6','7','8'
		160	DB 0E6H ; '9'			160	DB 0E6H ; '9'
		161	;			161	;
		162	*****			162	*****
		163	;			163	;
		164	;			164	;
		165	;These part of the program reads an analog			165	;These part of the program reads an analog
			;input-channel.				;input-channel.
		166	;Displaying is done on the LED-display			166	;Displaying is done on the LED-display
		167	;On odd-seconds the channel number will be			167	;On odd-seconds the channel number will be
			;displayed.				;displayed.
		168	;On even-seconds the analog value of this channel is			168	;On even-seconds the analog value of this channel is
			;displayed				;displayed
		169	;Then the next channel is displayed.			169	;Then the next channel is displayed.
		170	;			170	;
00A6: EB		171	ADCON: MOV A,R3 ;Get seconds	00A6: EB		171	ADCON: MOV A,R3 ;Get seconds
00A7: 13		172	RRC A ;lsb to carry	00A7: 13		172	RRC A ;lsb to carry
00A8: 503C		173	JNC NEW_MEAS ;Even seconds; do a	00A8: 503C		173	JNC NEW_MEAS ;Even seconds; do a
			;measurement on the current				;measurement on the current
			;channel				;channel
		174	;			174	;
		175	;Display and/or update channel			175	;Display and/or update channel
00AA: 33		176	RLC A ;Restore accu	00AA: 33		176	RLC A ;Restore accu
00AB: B51402	R	177	CJNE A,PREVIOUS,NEW_CH ;If new seconds,	00AB: B51402	R	177	CJNE A,PREVIOUS,NEW_CH ;If new seconds,
			;update channel number				;update channel number
00AE: 800A		178	JMP DISP_CH	00AE: 800A		178	JMP DISP_CH
00B0: 0515	R	179	NEW_CH: INC CHANNEL	00B0: 0515	R	179	NEW_CH: INC CHANNEL
00B2: E515	R	180	MOV A,CHANNEL ;If channel=4 then	00B2: E515	R	180	MOV A,CHANNEL ;If channel=4 then
			;channel:=0				;channel:=0
00B4: B40403		181	CJNE A,#04,DISP_CH	00B4: B40403		181	CJNE A,#04,DISP_CH
00B7: 751500	R	182	MOV CHANNEL,#00	00B7: 751500	R	182	MOV CHANNEL,#00
00BA: 8B14	R	183	DISP_CH:MOV PREVIOUS,R3 ;Update previous seconds	00BA: 8B14	R	183	DISP_CH:MOV PREVIOUS,R3 ;Update previous seconds
00BC: E515	R	184	MOV A,CHANNEL ;Get segment value of	00BC: E515	R	184	MOV A,CHANNEL ;Get segment value of
			;channel				;channel
00BE: 900193	R	185	MOV DPTR,#LED_TAB	00BE: 900193	R	185	MOV DPTR,#LED_TAB
00C1: 93		186	MOVC A,@A+DPTR	00C1: 93		186	MOVC A,@A+DPTR
		187	;			187	;
00C2: 7800	R	188	MOV R0,#TABLE ;Fill table with I2C data	00C2: 7800	R	188	MOV R0,#TABLE ;Fill table with I2C data

Interfacing the PCD8584 I²C-bus controller to 80C51 family microcontrollers

AN425

ASM51 TSW ASSEMBLER Demo program for PCD8584 I²C-routines

LOC	OBJ	LINE	SOURCE	LINE	OBJ
00C4: 7600		189	MOV @R0,#00 ;SAA1064 instruction byte	238	
00C6: 08		190	INC R0	239	
00C7: 7677		191	MOV @R0,#77H ;SAA1064 control byte	240	
00C9: 08		192	INC R0	241	
00CA: F6		193	MOV @R0,A ;Channel number	242	
00CB: E4		194	CLR A	243	
00CC: 08		195	INC R0	244	
00CD: F6		196	MOV @R0,A ;Second digit	245	
00CE: 08		197	INC R0	246	
00CF: F6		198	MOV @R0,A ;Third digit	247	
00D0: 08		199	INC R0	248	
00D1: F6		200	MOV @R0,A ;Fourth byte	249	
		201	;	250	
00D2: D200	R	202	SETB DIR ;I2C transmission of channel	251	
			;number	252	
00D4: 750000	R	203	MOV BASE,#TABLE	253	
00D7: 750006	R	204	MOV NR_BYTES,#06H	254	
00DA: 750076	R	205	MOV SLAVE,#SAA1064W	255	
00DD: 120000	R	206	CALL START	256	
		207	;	257	
00E0: 3000FD	R	208	JNB I2C_END,FIN_5	258	
00E3: 020027	R	209	JMP REPEAT ; Repeat clock and AD cycle	259	
			; again	260	
		210	;	261	
		211	;	262	
		212	;Measure and display the value of an AD-channel	263	
00E6: 120108	R	213	NEW_MEAS: CALL AD_VAL ;Do measurement	264	
		214	;Wait till values are available	265	
00E9: 3000FD	R	215	FIN_6: JNB I2C_END,FIN_6	266	
		216	;Relevant byte in TABLE+1. Transfer to AN_VAL	267	
00EC: 7801	R	217	MOV R0,#TABLE+1	268	
00EE: 8616	R	218	MOV AN_VAL,@R0	269	
00F0: E516	R	219	MOV A,AN_VAL ;Channel value in accu for	270	
			;conversion	271	
		220	;AN_VAL is converted to BCD value of the measured	272	
			;voltage.	273	
		221	;Input value for CONVERT in accu.	274	
		222	;Address for MSByte in R1	275	
00F2: 7917	R	223	MOV R1,#CONVAL	276	
00F4: 120154	R	224	CALL CONVERT	277	
		225	;Convert 3 bytes of CONVAL to LED-segments	278	
00F7: 900193	R	226	MOV DPTR,#LED_TAB ;Base of segment table	279	
00FA: 7817	R	227	MOV R0,#CONVAL	280	
00FC: 12018A	R	228	CALL SEG_LOOP	281	
		229	;Display value of channel to LED display	282	
00FF: 12012C	R	230	CALL LED_DISP	283	
0102: 3000FD	R	231	FIN_8: JNB I2C_END,FIN_8 ;Wait till I2C	284	
			;transmission is ended	285	
0105: 020027	R	232	JMP REPEAT ;Repeat clock and AD cycle	286	
		233	;	287	
		234	*****	288	
		235	;Routines used for AD converter.	289	
		236	;	290	
		237	;AIN reads an analog values from channel denoted by	291	
			;CHANNEL.	292	

Interfacing the PCD8584 I²C-bus controller to 80C51 family microcontrollers

AN425

ASM51 TSW ASSEMBLER Demo program for PCD8584 I²C-routines

LOC	OBJ	LINE	SOURCE	LOC	OBJ	LINE	SOURCE
		238	;Send controlbyte:				
0108: D200	R	239	AD_VAL: SETB DIR ;I2C transmission				
010A: 7800	R	240	MOV R0,#TABLE ;Define control word				
010C: A615	R	241	MOV @R0,CHANNEL				
010E: 750000	R	242	MOV BASE,#TABLE ;Set base at table				
0111: 750001	R	243	MOV NR_BYTES,#01H ;Number of bytes to be ;send				
0114: 75009E	R	244	MOV SLAVE,#PCF8591W ;Slave address PCF8591				
0117: 120000	R	245	CALL START ;Start transmission of ;controlword				
011A: 3000FD	R	246	FIN_7: JNB I2C_END,FIN_7 ;Wait until transmission is ;finished				
		247	;Read 2 data bytes from AD-converter				
		248	;First data byte is from previous conversion and not ;relevant				
011D: C200	R	250	CLR DIR ;I2C reception				
011F: 750000	R	251	MOV BASE,#TABLE ;Bytes must be stored in ;TABLE				
0122: 750002	R	252	MOV NR_BYTES,#02H; Receive 3 bytes				
0125: 75009F	R	253	MOV SLAVE,#PCF8591R ;Slave address PCF8591				
0128: 120000	R	254	CALL START				
012B: 22		255	RET				
		256	;				
		257	;LED_DISP displays the data of 3 bytes from address ;CONVAL				
012C:		258	LED_DISP:				
012C: 431780	R	259	ORL CONVAL,#80H ;Set decimal point				
012F: 7800	R	260	MOV R0,#TABLE				
0131: 7917	R	261	MOV R1,#CONVAL				
0133: 7600		262	MOV @R0,#00 ;SAA1064 instruction byte				
0135: 08		263	INC R0				
0136: 7677		264	MOV @R0,#01110111B ;SAA1064 control byte				
0138: 08		265	INC R0				
0139: 7600		266	MOV @R0,#00 ;First LED digit				
013B: 08		267	INC R0				
013C: 120185	R	268	CALL GETBY ;Second digit				
013F: 120185	R	269	CALL GETBY ;Third digit				
0142: 120185	R	270	CALL GETBY ;Fourth digit				
0145: D200	R	271	SETB DIR ;I2C transmission				
0147: 750000	R	272	MOV BASE,#TABLE				
014A: 750006	R	273	MOV NR_BYTES,#06				
014D: 750076	R	274	MOV SLAVE,#01110110B				
0150: 120000	R	275	CALL START ;Start I2C transmission				
0153: 22		276	RET				
		277	;				
		278	;CONVERT calculates the voltage of the analog value.				
		279	;Analog value must be in accu				
		280	;BCD result (3 bytes) is stored from address stored ;in R1				
		281	;Calculation: AN_VAL*(5/256)				
0154: 75F005		282	CONVERT:MOV B,#05				
0157: A4		283	MUL AB				
		284	;b2..b0 of reg. B : 2E+2..2E0				
		285	;b7..b0 of accu : 2E-1..2E-8				
0158: A7F0		286	MOV @R1,B ;Store MSB (10E0-units)				
015A: 09		287	INC R1				

Interfacing the PCD8584 I²C-bus controller to 80C51 family microcontrollers

AN425

ASM51 TSW ASSEMBLER Demo program for PCD8584 I2C-routines

LOC	OBJ	LINE	SOURCE
015B: 7700		288	MOV @R1,#00 ;Calculate 10E-1 unit ;(10E-1 is 19h)
015D: B41C02		289	TEN_CH: CJNE A,#19H+03H,V1 ;Check if accu <= 0.11
0160: 8002		290	JMP TENS ;accu=0.11; update tens
0162: 4006		291	V1: JC NX_CON ;accu<0.11; update hundreds
0164: C3		292	TENS: CLR C ;Calculate new value
0165: 9419		293	SUBB A,#19H
0167: 07		294	INC @R1 ;Update BCD byte
0168: 80F3		295	JMP TEN_CH
		296	;Correction may be necessary. With 8 bits '0.1' is ;in fact 0.0976.
		297	;A digit of '0A' may appear. Correct this by ;decrementing the digit.
		298	;The intermediate result result must be corrected ;with 10*(0.1-0.0976)
		299	;This is 06H
016A: B70A03		300	NX_CON: CJNE @R1,#0AH,PROC_CON ; If digit is '0A' ;then correct
016D: 17		301	DEC @R1
016E: 2419		302	ADD A,#19H
0170: 09		303	PROC_CON:INC R1
0171: 7700		304	MOV @R1,#00 ;Calculate 10E-2 units
0173: B40302		305	HUND: CJNE A,#03H,V2 ;Check if accu <= 10E-2
0176: 8002		306	JMP HUNS ;accu=10E-2; update hundreds
0178: 4006		307	V2: JC FINISH ;accu<10E-2; conversion ;finished
017A: C3		308	HUNS: CLR C ;Calculate new value
017B: 9403		309	SUBB A,#03H
017D: 07		310	INC @R1 ;Update BCD byte
017E: 80F3		311	JMP HUND
0180: B70A01		312	FINISH: CJNE @R1,#0AH,FIN ;Check if result is '0A'. ;Then correct.
0183: 17		313	DEC @R1
0184: 22		314	FIN: RET
		315	;
		316	;CALLBY transfers byte from @R1 to @R0
0185: E7		317	GETBY: MOV A,@R1
0186: F6		318	MOV @R0,A
0187: 08		319	INC R0
0188: 09		320	INC R1
0189: 22		321	RET
		322	;
		323	;SEG_LOOP converts 3 values to segment values.
		324	;R0 contains address of source and destination
		325	;DPTR contains base of table
018A: 7903		326	SEG_LOOP: MOV R1,#03 ;Loop counter
018C: E6		327	INLOOP: MOV A,@R0 ;Get value to be displayed
018D: 93		328	MOVC A,@A+DPTR ;Get segment value from ;table
018E: F6		329	MOV @R0,A ;Store segment data
018F: 08		330	INC R0
0190: D9FA		331	DJNZ R1,INLOOP
0192: 22		332	RET
		333	;
		334	;

Interfacing the PCD8584 I²C-bus controller to 80C51 family microcontrollers

AN425

ASM51 TSW ASSEMBLER Demo program for PCD8584 I2C-routines Demo program for PCD8584 I2C-routines

```

LOC   OBJ           LINE   SOURCE
-----
0193: 7D483E        335   ;LED_TAB is conversion table for BCD to LED segments
0196: 6E4B67        336   LED_TAB:
0199: 734C7F        337       DB 7DH,48H,3EH ; '0','1','2'
019C: 4F             338       DB 6EH,4BH,67H ; '3','4','5'
019C: 4F             339       DB 73H,4CH,7FH ; '6','7','8'
019C: 4F             340       DB 4FH ; '9'
019C: 4F             341   ;
019C: 4F             342   ;*****
019C: 4F             343   ;
019C: 4F             344   RSEG RAMTAB ; With 2 necessary.
0000: R             345   TABLE: DS 10
000A:              346   ;
000A:              347   END

```

Using the 8XC751/752 in multimaster I²C applications AN430

INTRODUCTION

The Philips Semiconductors 83C751/87C751 offers the advantages of the 80C51 architecture in a small package and at a low cost. It combines the benefits of a high performance microcontroller with on-board hardware supporting the Inter Integrated Circuit (I²C) bus interface.

The Inter IC (I²C) bus developed by Philips allows integrated circuits to communicate directly with each other via a simple bidirectional 2-wire bus. The comprehensive family of CMOS and bipolar ICs incorporating the on-chip I²C interface offers many advantages to designers of digital control for industrial, consumer and telecommunications equipment.

Interfacing the devices in an I²C based system is very simple as they connect directly to the two bus lines: a serial data line (SDA) and a serial clock line (SCL). System design can rapidly progress from block diagram to final schematics, as there is no need to design bus interfaces. In addition, functional blocks on the block diagram correspond to actual ICs. A prototype system or a final product version can be easily modified or upgraded by 'clipping' or 'unclipping' ICs to or from the bus. The simplicity of designing with the I²C bus does not reduce its effectiveness: it is a reliable, multimaster bus with integrated addressing and data-transfer protocols. The I²C-bus compatible ICs give cost reduction benefits through smaller IC packages and a minimization of PCB traces and glue logic.

The availability of microcontrollers, like the 83C751, with on-board I²C interface is a very powerful tool for system designers. The integrated protocols allow systems to be completely software defined. Software development time of different products can be reduced by assembling a library of re-usable software modules. In addition, the multimaster capability allows rapid testing and alignment of end-products via external connections to an assembly-line computer.

The mask programmable 83C751 and its EPROM version, 87C751, can operate as a master or a slave device on the I²C small area network. In addition to the efficient interface to the dedicated function ICs in the I²C family the on-board interface facilitates I/O and RAM expansion, access to EEPROM, and processor-to-processor communications.

The 83C752 and its EPROM version, 87C752, are essentially the 83C751/87C751 with the addition of a five channel multiplexed 8-bit A/D converter and an 8-bit PWM output. As the I²C bus interface is identical, the programming example and the discussion relates to both processors. The multimaster capability of the I²C bus allows easy integration and expansion of relatively complex systems, in which different devices can independently initiate data transfers. Integration of a multimaster system is easy as a Master on the bus does not have to coordinate its data transfer with other potential Master devices—arbitration and synchronization are taken care of by the

hardware and bus protocols. Expanding a system with a new device is trivial—it is "clipped" onto the two serial bus lines, and the new device may act as a Master without any modification to the other devices (see Figure 1). Microcontrollers like the 83C751/752 on the I²C bus are extremely powerful, as they can be programmed to be both Masters and Slaves in the same system. This way the microcontroller may initiate communication on the bus, and when requested, will respond to a data transfer request by another device.

In this Application Note we shall discuss the most important technical features of the I²C bus and describe the special I²C hardware interface of the 83C751/752. We shall demonstrate with an example how the microcontroller can be programmed for a multimaster environment. The communications routines of the example are quite general, and can be ported to many applications—so we shall discuss in detail the software interface to these routines.

The description of the 83C751 I²C interface hardware and part of the general discussion of the I²C bus is similar to Application Note AN422 which dealt with the microcontroller in a single-master environment. Most of the added discussions relate to the multimaster aspects of the bus. Additional information for the I²C bus and the 83C751/752 Microcontroller can be found in the Philips Semiconductors Microcontroller Data Handbook (IC20).

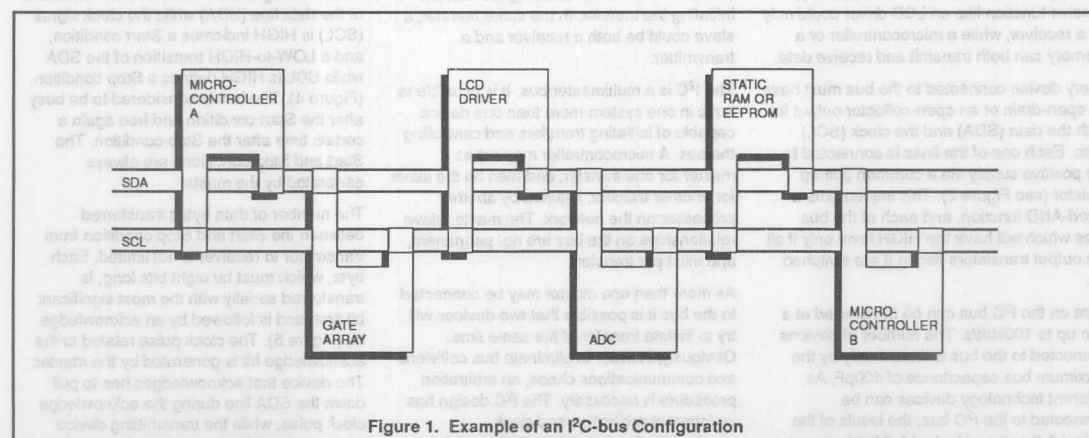


Figure 1. Example of an I²C-bus Configuration

Using the 8XC751/752 in multimaster I²C applications AN430

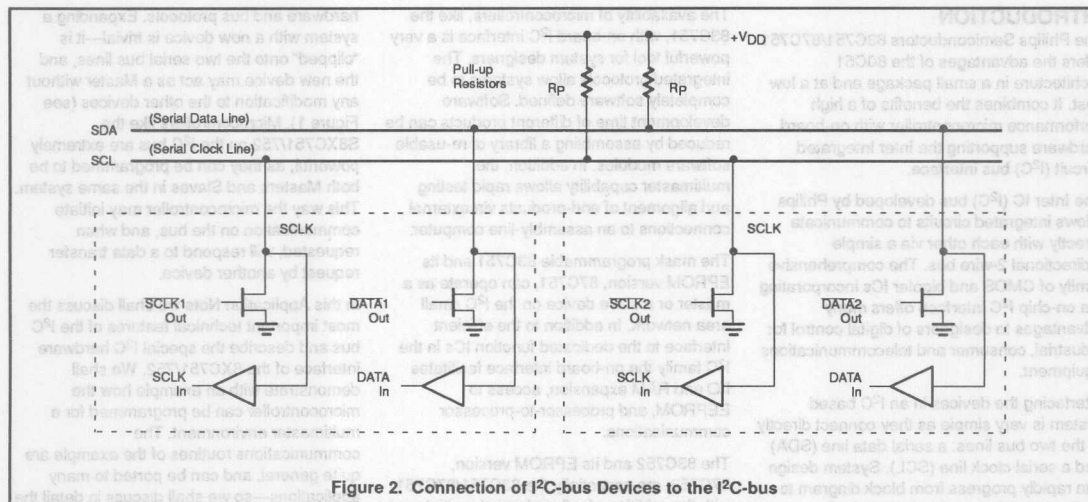


Figure 2. Connection of I²C-bus Devices to the I²C-bus

THE I²C BUS

The two lines of the I²C bus are a serial data line (SDA) and a serial clock line (SCL). A typical system configuration is shown in Figure 2. Each device is recognized by a unique address—whether it is a microcomputer, LCD driver, memory or keyboard interface—and can operate as either a transmitter or a receiver, depending on the function of the device. A device generating a message or data is a transmitter, and a device receiving the message or data is a receiver. Obviously, a passive function like an LCD driver could only be a receiver, while a microcontroller or a memory can both transmit and receive data.

Every device connected to the bus must have an open-drain or an open-collector output for both the data (SDA) and the clock (SCL) lines. Each one of the lines is connected to the positive supply via a common pull-up resistor (see Figure 2). This implements a wired-AND function, and each of the bus lines which will have the HIGH level only if all the output transistors tied to it are switched off.

Data on the I²C bus can be transferred at a rate up to 100kbit/s. The number of devices connected to the bus is limited only by the maximum bus capacitance of 400pF. As different technology devices can be connected to the I²C bus, the levels of the logical 0 (Low) and logical 1 (High) are not fixed and depend on the appropriate level of V_{DD}.

MASTERS AND SLAVES

When a data transfer takes place on the bus, a device can be either a master or a slave. The device which initiates the transfer, and generates the clock signals for this transfer is the master. At that time any device addressed is considered a slave. It is important to note that a master could be either a transmitter or a receiver: a master microcontroller may send data to a RAM acting as a transmitter, and then interrogate the RAM for its contents acting as a receiver—in both cases being the master initiating the transfer. In the same manner, a slave could be both a receiver and a transmitter.

The I²C is a multimaster bus. It is possible to have in one system more than one device capable of initiating transfers and controlling the bus. A microcontroller may act as a master for one transfer, and then be the slave for another transfer, initiated by another processor on the network. The master/slave relationships on the bus are not permanent, and exist per transfer.

As more than one master may be connected to the bus it is possible that two devices will try to initiate transfer at the same time. Obviously, in order to eliminate bus collisions and communications chaos, an arbitration procedure is necessary. The I²C design has an inherent arbitration and clock synchronization procedure relying on the wired-AND connection of the devices on the bus. In a typical multimaster system, a

microcontroller program should allow it to gracefully switch between master and slave modes and preserve data integrity upon loss of arbitration.

DATA TRANSFERS

One data bit is transferred during each clock pulse (Figure 3). The data on the SDA line must remain stable during the HIGH period of the clock pulse in order to be valid. Changes in the data line at this time will be interpreted as control signals. A HIGH-to-LOW transition of the data line (SDA) while the clock signal (SCL) is HIGH indicates a Start condition, and a LOW-to-HIGH transition of the SDA while SCL is HIGH defines a Stop condition (Figure 4). The bus is considered to be busy after the Start condition and free again a certain time after the Stop condition. The Start and Stop conditions are always generated by the master.

The number of data bytes transferred between the Start and Stop condition from transmitter to receiver is not limited. Each byte, which must be eight bits long, is transferred serially with the most significant bit first, and is followed by an acknowledge bit (Figure 5). The clock pulse related to the acknowledge bit is generated by the master. The device that acknowledges has to pull down the SDA line during the acknowledge clock pulse, while the transmitting device releases the SDA line (HIGH) during this pulse (Figure 6).

Using the 8XC751/752 in multimaster I²C applications

AN430

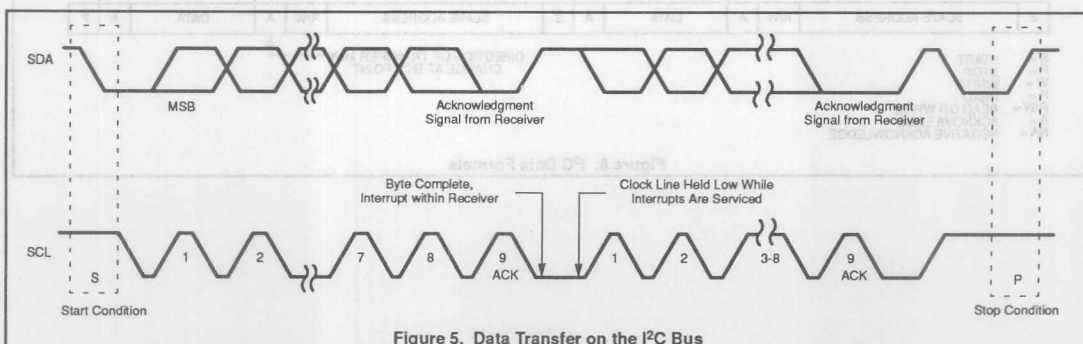
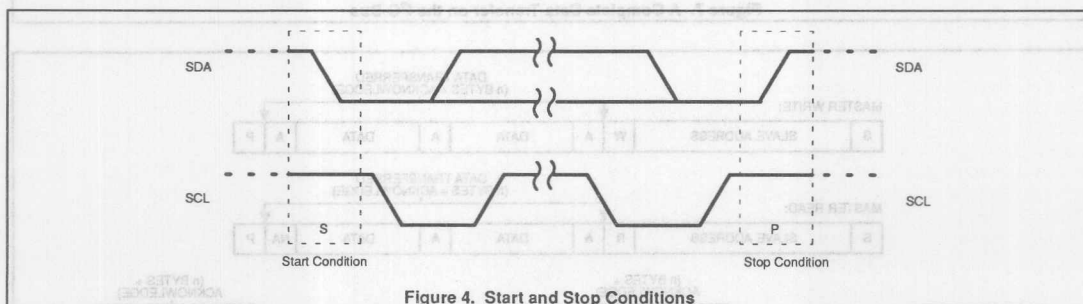
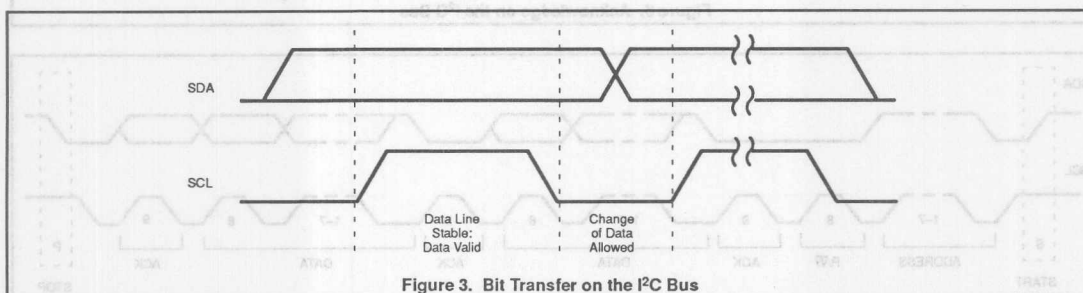
A slave receiver must generate an acknowledge after the reception of each byte, and a master must generate one after the reception of each byte clocked out of the slave transmitter. If a receiving device cannot receive the data byte immediately, it can force the transmitter into a wait state by holding the clock line (SCL) LOW. When designing a system it is necessary to take into account cases when acknowledge is not received. This happens, for example, when the addressed device is busy in a real time operation. In such a case the master, after an appropriate "time-out", should abort the

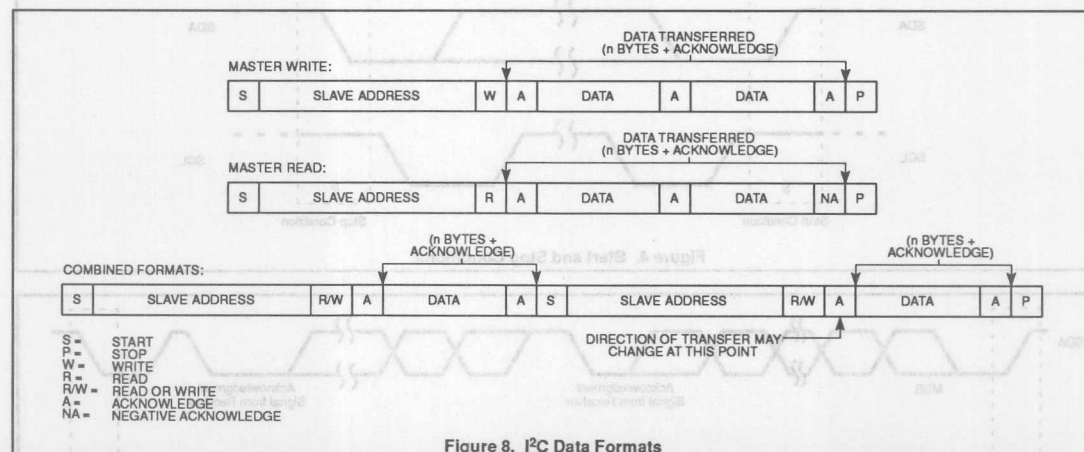
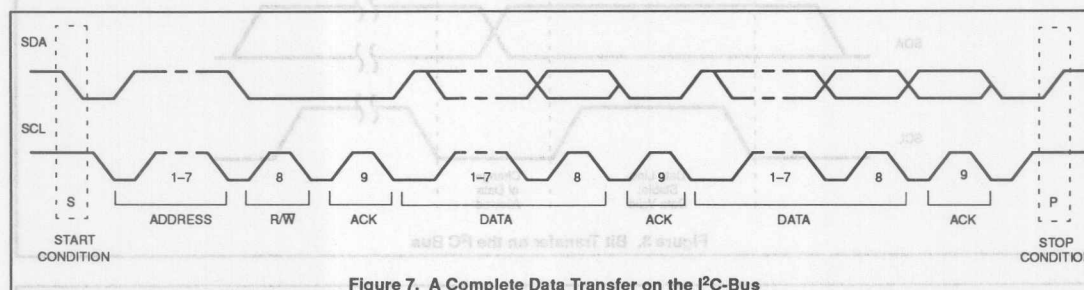
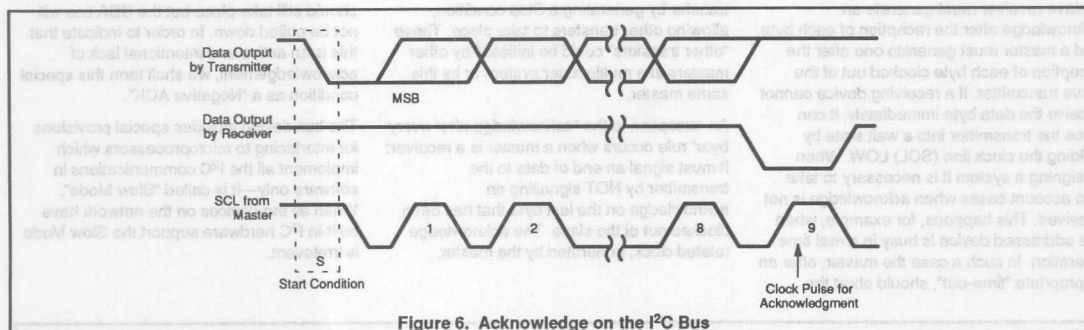
transfer by generating a Stop condition, allowing other transfers to take place. These "other transfers" could be initiated by other masters in a multimaster system or by this same master.

An exception to the "acknowledge after every byte" rule occurs when a master is a receiver: it must signal an end of data to the transmitter by NOT signalling an acknowledge on the last byte that has been clocked out of the slave. The acknowledge related clock, generated by the master,

should still take place but the SDA line will not be pulled down. In order to indicate that this is an active and intentional lack of acknowledgement, we shall term this special condition as a "Negative ACK".

The bus design includes special provisions for interfacing to microprocessors which implement all the I²C communications in software only—it is called "Slow Mode". When all the devices on the network have built-in I²C hardware support the Slow Mode is irrelevant.



Using the 8XC751/752 in multimaster I²C applications AN430

Using the 8XC751/752 in multimaster I²C applications AN430

ADDRESSING AND TRANSFER FORMATS

Each device on the bus has its own unique address. Before any data is transmitted on the bus, the master transmits on the bus the address of the slave of this transaction. A well-behaved slave, if it exists on the network, should of course acknowledge the master's addressing. The addressing is done with the first byte transmitted by the master after the Start condition.

An address on the network is seven bits long, appearing as the most significant bits of the address byte. The last bit is a direction (R/W) bit. A zero indicates that the master is transmitting (WRITE) and a one indicates that the master requests data (READ). A complete data transfer, comprised of an address byte indicating a WRITE and two data bytes is shown in Figure 7.

When an address is sent, each device in the system compares the first seven bits after the Start with its own address. If there is a match, the device will consider itself addressed by the master and will send an acknowledge. The device could also determine if in this transaction it is assigned the role of a slave receiver or slave transmitter, depending on the R/W bit.

Each node of the I²C network has a unique seven bit address. The address of a microcontroller is, of course, fully programmable, while peripheral devices usually have fixed and programmable address portions. In addition to the "standard" addressing discussed here, the I²C bus protocol allows for "general call" addressing and interfacing to CBUS devices.

When the master is communicating with one device only, data transfers follow the format of Figure 8 where the R/W bit could indicate

either direction. After completing the transfer and issuing a Stop condition, if a master would like to address some other device on the network, it could start another transaction by issuing a new Start.

Another way for a master to communicate with several different devices would be by using a "repeated start". After the last byte of the transaction was transferred, including its acknowledge (or Negative ACK), the master issues again a Start, followed by address byte and data, without effecting a Stop. The master may communicate with a number of different devices, combining READS and WRITES. Only after the transfer with the last slave took place, the master issues a Stop and releases the bus. Possible data formats are demonstrated in Figure 8. Note that the repeated start allows for both change of a slave and a change of direction, without releasing the bus. We shall see later on that the change of direction feature can come in handy even when dealing with a single device.

In a single master system the repeated start mechanism is more efficient than terminating each transfer with a Stop and starting again. In a multimaster environment the determination of which format is more efficient could be more complicated, as when a master is using repeated starts it occupies the bus for a long time and prevents other devices from initiating transfers.

USE OF SUB-ADDRESSES

For some ICs on the I²C bus the device address alone is not sufficient for effective communications and a mechanism for addressing the internals of the device is necessary. A typical example is addressing memories, when we want to access a specific

word inside the device or a sequence of memory locations starting at a specific internal address.

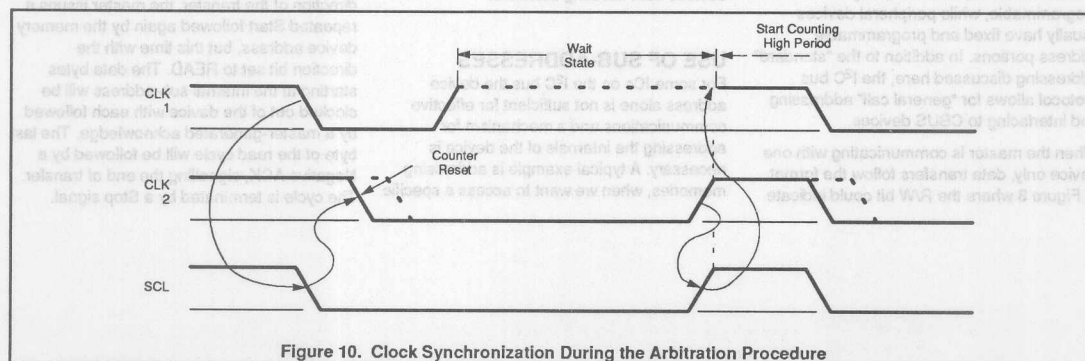
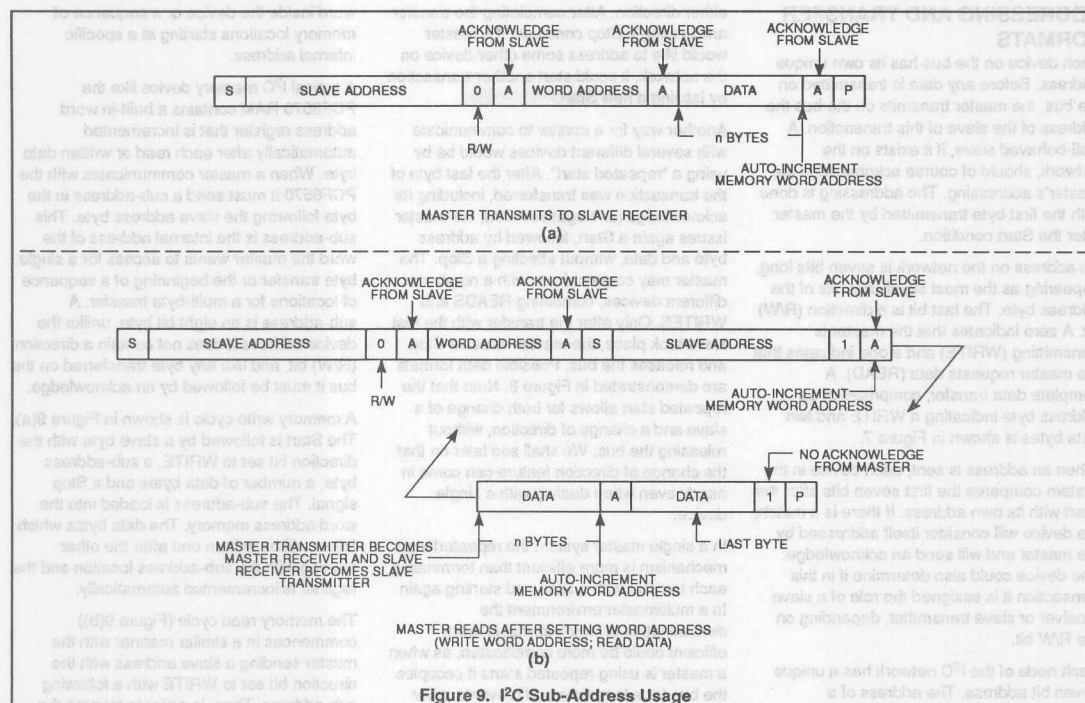
A typical I²C memory device like the PCF8570 RAM contains a built-in word address register that is incremented automatically after each read or written data byte. When a master communicates with the PCF8570 it must send a sub-address in the byte following the slave address byte. This sub-address is the internal address of the word the master wants to access for a single byte transfer or the beginning of a sequence of locations for a multi-byte transfer. A sub-address is an eight bit byte, unlike the device address it does not contain a direction (R/W) bit, and like any byte transferred on the bus it must be followed by an acknowledge.

A memory write cycle is shown in Figure 9(a). The Start is followed by a slave byte with the direction bit set to WRITE, a sub-address byte, a number of data bytes and a Stop signal. The sub-address is loaded into the word address memory. The data bytes which follow will be written one after the other starting with the sub-address location and the register is incremented automatically.

The memory read cycle (Figure 9(b)) commences in a similar manner with the master sending a slave address with the direction bit set to WRITE with a following sub-address. Then, in order to reverse the direction of the transfer, the master issues a repeated Start followed again by the memory device address, but this time with the direction bit set to READ. The data bytes starting at the internal sub-address will be clocked out of the device with each followed by a master-generated acknowledge. The last byte of the read cycle will be followed by a Negative ACK, signalling the end of transfer. The cycle is terminated by a Stop signal.

Using the 8XC751/752 in multimaster I²C applications

AN430



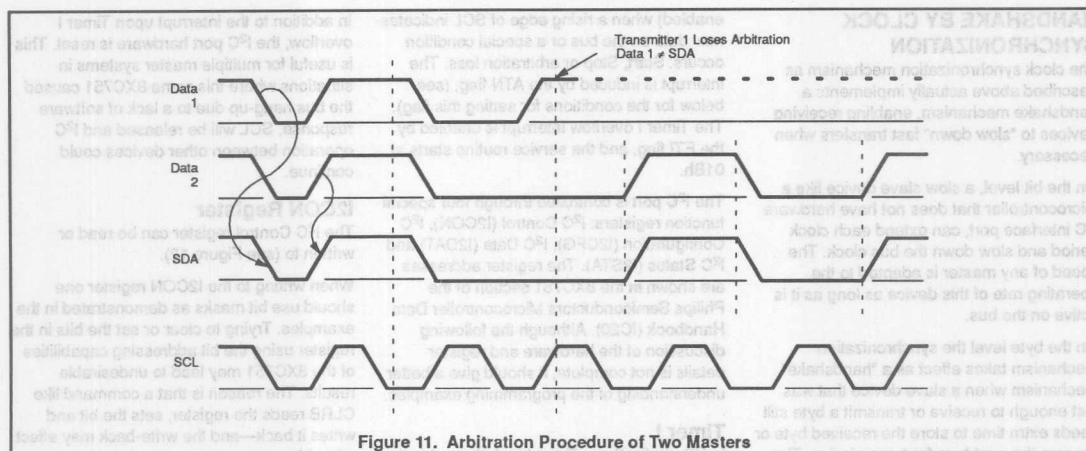
Using the 8XC751/752 in multimaster I²C applications AN430

Figure 11. Arbitration Procedure of Two Masters

ARBITRATION IN A MULTIMASTER SYSTEM

The decision about which master has control over the I²C bus is based solely on the address and data sent by competing masters, and there is no central master or any order of device priority on the bus. Any device connected to the I²C bus is allowed to become a master, but devices are not supposed to "steal" the bus from other devices when a transfer is in process. If a device wishing to be a Master is aware that a transaction (initiated by another master) is taking place, it will wait until the transfer is concluded with a Stop condition on the bus—and only then try to seize it by sending its own Start. It is possible, however, that two or more masters may want to start a transfer at exactly the same moment. A scenario that may happen quite frequently in a loaded system: two devices are waiting for a long transaction to be completed, and simultaneously try to get the bus when detecting the Stop condition. An arbitration procedure synchronizes the different clocks, ensuring that the data is not corrupted, and causes all masters except one to withdraw from the bus, so only one master will control the transfer. This procedure applies only when masters initiate transfers simultaneously.

The clock synchronization, illustrated in Figure 10, ensures that only one defined clock is generated on the bus. It occurs naturally, as a result of the wired-AND property of the SCL line. Suppose two masters want to initiate a transfer on the bus.

Clk1 and Clk2 in Figure 10 illustrate the desired clock outputs of each device, which would actually occur on the bus if each were the only master. The SCL waveform is the resulting wired-AND of the two clocks. The device that pulls the SCL down first will succeed. The other masters continuously monitor the clock line, and reset their internal clock counter to start counting their own Low clock period. This way, the first falling edge will synchronize all clock generators to the beginning of the Low time.

Once a device clock has gone Low it will hold the SCL line in this state until its internal clock High state is reached, and then will release the line. The Low to High change in this device will not change the state of the SCL line if another device, which is still within its Low period, is pulling down the line. This way, SCL will be held Low by the device with the longest Low period. A master that has finished its Low time earlier will enter a wait state until SCL is released by the slowest master and goes high. Upon the rising edge of SCL all masters start counting their High period, the first device to complete its High period will pull the SCL Low. In this way a single, synchronized clock is generated on the bus where the rising edge is being defined by the slowest master and the falling edge by the fastest master.

Arbitration between masters takes place on the SDA line. A master which tries to transmit a High while another device transmits a Low will withdraw, shutting off its data output stage and not interfering with the transfer until a Stop condition is detected. Due to the

wired-AND property of the SDA line, a device "knows" that it lost arbitration by the fact that the Low SDA is different than its desired High output. Arbitration starts by comparing the address bits. When masters transmit different addresses the one transmitting the address with the lowest binary value wins. If all masters in arbitration transmit to the same address, arbitration continues into the comparison of data. Figure 11 illustrates the arbitration process between two masters.

By definition, the transfer that forces the wired-AND result is the one that wins the arbitration, so the address and data of a winning device are not corrupted and no information is lost in the arbitration process. A master losing arbitration may generate clock pulses until the end of the byte. Thus it may affect the clock speed, but not the data on the bus.

If a master loses arbitration during the addressing stage it is possible that the winning master is trying to address it. In an efficient design, the losing master should switch immediately to its slave receiver mode, receive the data transmitted and acknowledge it—otherwise the message will have to be re-transmitted or is lost. A well designed master will take into account "illegal" protocol situations and will determine that it lost arbitration when it detects a Stop or a Start which are not synchronized with its own transmission. Electrical interference or a malfunctioning device may cause such a situation which actually corrupts the message transfer.

Using the 8XC751/752 in multimaster I²C applications AN430

HANDSHAKE BY CLOCK SYNCHRONIZATION

The clock synchronization mechanism as described above actually implements a handshake mechanism, enabling receiving devices to "slow down" fast transfers when necessary.

On the bit level, a slow slave device like a microcontroller that does not have hardware I²C interface port, can extend each clock period and slow down the bus clock. The speed of any master is adapted to the operating rate of this device as long as it is active on the bus.

On the byte level the synchronization mechanism takes effect as a "handshake" mechanism when a slave device that was fast enough to receive or transmit a byte still needs extra time to store the received byte or prepare the next byte for transmission. The slave can hold the SCL line low after the reception and acknowledge of a byte, thus forcing the Master into a wait state—until the slave is ready for the next transfer.

8XC751 I²C HARDWARE

The on-chip I²C bus hardware support of the 8XC751 allows operation on the bus at full speed and simplifies the software needed for effective communications on the network. The hardware activates and monitors the SDA and SCL lines, performs the necessary arbitration and framing error checks, and takes care of clock stretching and synchronization. The hardware support includes a bus timeout timer, called Timer I. The hardware is synchronized to the software either through polled loops or interrupts.

Two of the port 0 pins are multi-functional. When the I²C is active, the pin associated with P0.0 functions as SCL, and the pin associated with P0.1 functions as SDA. These pins have an open drain output.

Two of the five interrupt sources may be used for I²C support. The I²C interrupt is enabled by the EI2 flag of the interrupt enable register, and its service routine should start at address 023h. An I²C interrupt is usually requested (if

enabled) when a rising edge of SCL indicates new data on the bus or a special condition occurs: Start, Stop or arbitration loss. The interrupt is induced by the ATN flag, (see below for the conditions for setting this flag). The Timer I overflow interrupt is enabled by the ETI flag, and the service routine starts at 01Bh.

The I²C port is controlled through four special function registers: I²C Control (I2CON), I²C Configuration (I2CFG), I²C Data (I2DAT) and I²C Status (I2STA). The register addresses are shown in the 8XC751 section of the Philips Semiconductors Microcontroller Data Handbook (IC20). Although the following discussion of the hardware and register details is not complete, it should give a better understanding of the programming examples.

Timer I

In I²C applications, Timer I is dedicated to the port timing generation and bus monitoring. In non-I²C applications, it is available for use as a fixed rate timer.

For the bus monitoring function, Timer I is being used as a "watchdog timer" for bus hang-ups. It creates an interrupt when the SCL line stays in one state for an extended period of time between a Start condition and a following Stop condition. SCL "stuck low" indicates a faulty master or slave. SCL "stuck high" may mean a faulty device or that noise induced into the I²C caused all masters to withdraw from the I²C arbitration.

The time-out interval of Timer I is fixed: it carries out and interrupts (if enabled) when about 1024 machine cycles have elapsed since a change on SCL within a frame. In other words, whenever I²C is active we let Timer I run, but clear it whenever a frame is not in progress (reset or Stop occurred more recently than the last Start condition) or SCL changes within a frame. (Note: we wrote "about 1024 machine cycles" for the sake of accuracy—this number may slightly change according to the setting of the CT0 and CT1 bits mentioned below. In any case, the exact number of cycles for a time out does not have any practical significance).

In addition to the interrupt upon Timer I overflow, the I²C port hardware is reset. This is useful for multiple master systems in situations where this same 8XC751 caused the bus hang-up due to a lack of software response. SCL will be released and I²C operation between other devices could continue.

I2CON Register

The I²C Control register can be read or written to (see Figure 12).

When writing to the I2CON register one should use bit masks as demonstrated in the examples. Trying to clear or set the bits in the register using the bit addressing capabilities of the 8XC751 may lead to undesirable results. The reason is that a command like CLRB reads the register, sets the bit and writes it back—and the write-back may affect other bits.

I2CFG Register

The configuration register is a read/write register (see Figure 13).

I2DAT Register

The I²C data register is a read/write register, where the msb represents the data received or data to be sent. The other seven bits are read as 0 (see Figure 14).

I2CSTA Register

The I²C STATUS Register is a read-only register reflecting the internal status of the I²C interface hardware (see Figure 15).

Transmit Active State

The transmit active state—Xmit Active—is an internal state in the I²C interface that is affected by the I²C registers as explained above. The I²C interface will only drive the SDA line low when Xmit Active is set. Xmit Active is set by writing the I2DAT register or by writing I2CON with XSTR = 1 or XSTP = 1. The ARL bit will be set to 1 only when Xmit Active is set—in such a case Xmit Active will be automatically reset upon ARL. Xmit Active is cleared by writing 1 to CXA at I2CON register or by reading the I2DAT register.

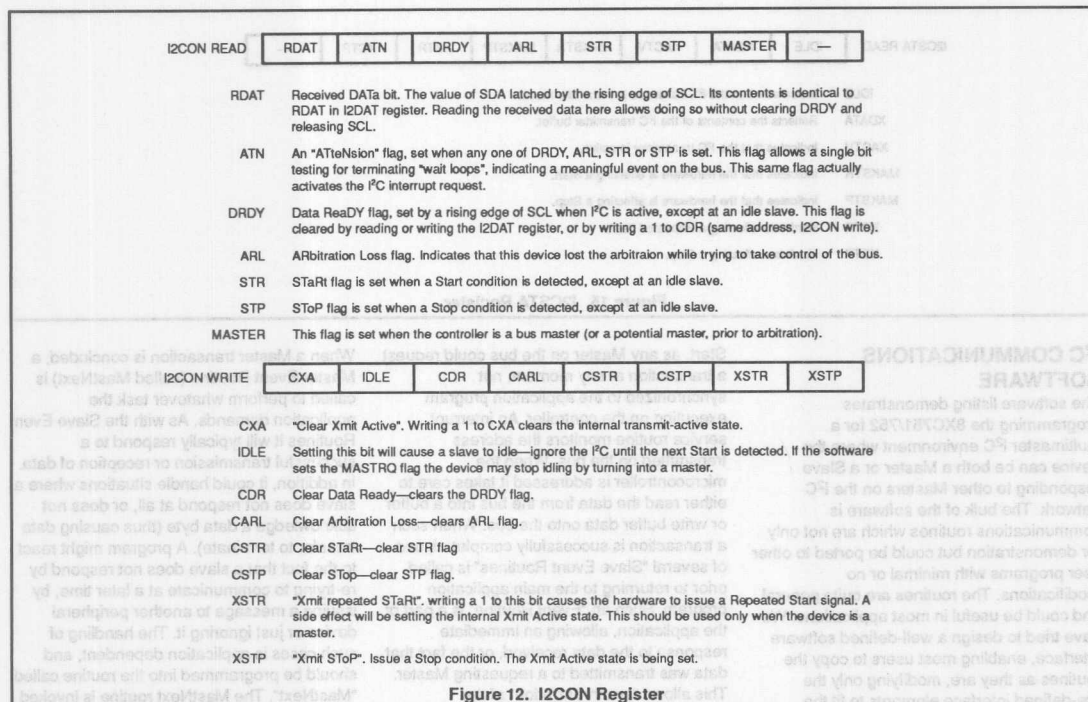
Using the 8XC751/752 in multimaster I²C applications AN430

Figure 12. I2CON Register

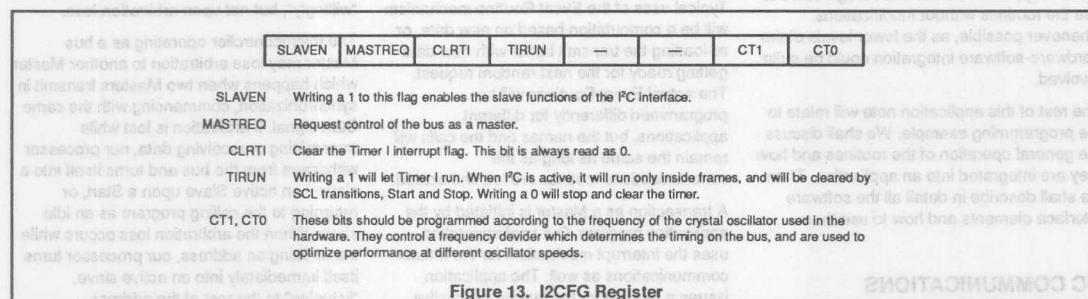


Figure 13. I2CFG Register

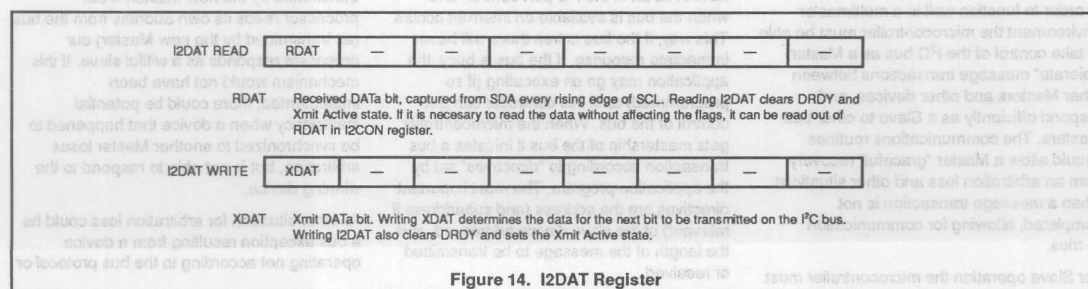
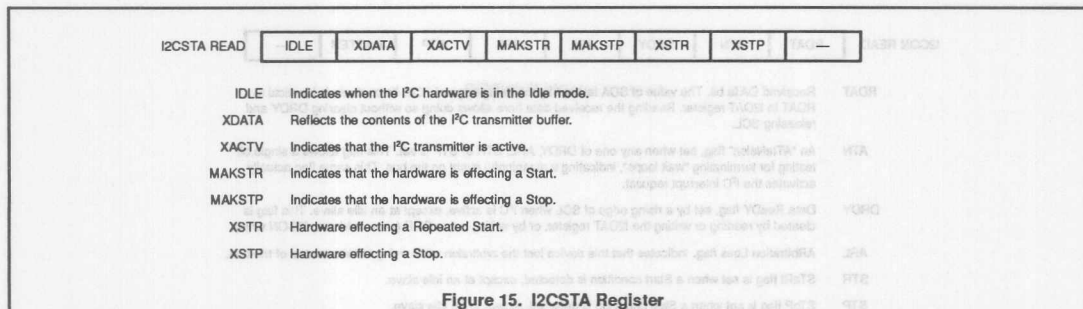


Figure 14. I2DAT Register

Using the 8XC751/752 in multimaster I²C applications AN430



I²C COMMUNICATIONS SOFTWARE

The software listing demonstrates programming the 8XC751/752 for a multimaster I²C environment where the device can be both a Master or a Slave responding to other Masters on the I²C network. The bulk of the software is communications routines which are not only for demonstration but could be ported to other user programs with minimal or no modifications. The routines are quite general and could be useful in most applications. We have tried to design a well-defined software interface, enabling most users to copy the routines as they are, modifying only the pre-defined interface elements to fit the specific applications. We encourage users to use the routines without modifications whenever possible, as the lower levels of the hardware-software integration could be quite involved.

The rest of this application note will relate to the programming example. We shall discuss the general operation of the routines and how they are integrated into an application. Then we shall describe in detail all the software interface elements and how to use them.

I²C COMMUNICATIONS ROUTINES—OVERVIEW

In order to function well in a multimaster environment the microcontroller must be able to take control of the I²C bus as a Master, "tolerate" message transactions between other Masters and other devices, and respond efficiently as a Slave to other bus Masters. The communications routines should allow a Master "graceful" recovery from an arbitration loss and other situations when a message transaction is not completed, allowing for communication re-tries.

For Slave operation the microcontroller must be interrupt driven relative to an I²C frame

Start, as any Master on the bus could request a transaction at any moment, not synchronized to the application program executing on the controller. An interrupt service routine monitors the address transmitted on the bus. When the microcontroller is addressed it takes care to either read the data from the bus into a buffer or write buffer data onto the bus. When such a transaction is successfully completed, one of several "Slave Event Routines" is called prior to returning to the main application program. Such an "Event Routine" is a part of the application, allowing an immediate response to the data received, or the fact that data was transmitted to a requesting Master. This allows "synchronization" of the application to a "slave" bus transaction. Typical uses of the Event Routine mechanism will be a computation based on new data, or re-loading the transmit buffer with new data getting ready for the next random request. The actual Event Routines will be programmed differently for different applications, but the names and the calls will remain the same as long as the communications routines are left unmodified.

A transaction as a Master is initiated by the application program. Our implementation uses the interrupt mechanism for the Master communications as well. The application issues a request for the bus by setting the MASTRQ bit of the I²C port control, and when the bus is available an interrupt occurs. This way, if the bus is free there will be an immediate response. If the bus is busy, the application may go on executing (if so programmed) until this controller can get control of the bus. When the microcontroller gets mastership of the bus it initiates a bus transaction according to "directives" set by the application program. The most important directives are the address (and subaddress if relevant) of the slave device addressed, and the length of the message to be transmitted or received.

When a Master transaction is concluded, a Master Event Routine (called MastNext) is called to perform whatever task the application demands. As with the Slave Event Routines it will typically respond to a successful transmission or reception of data. In addition, it could handle situations where a slave does not respond at all, or does not acknowledge a data byte (thus causing data transfer to terminate). A program might react to the fact that a slave does not respond by re-trying to communicate at a later time, by issuing a message to another peripheral device or just ignoring it. The handling of such cases is application dependent, and should be programmed into the routine called "MastNext". The MastNext routine is invoked when the Master terminates the transaction "willingly", but not upon arbitration loss.

The microcontroller operating as a bus Master may lose arbitration to another Master which happens when two Masters transmit in synchronization, commencing with the same Start signal. If arbitration is lost while transmitting or receiving data, our processor withdraws from the bus and turns itself into a slave—an active Slave upon a Start, or returning to the calling program as an idle slave. When the arbitration loss occurs while transmitting an address, our processor turns itself immediately into an active slave, "listening" to the rest of the address transmitted by the new Master. If our processor reads its own address from the bus (as transmitted by the new Master) our processor responds as a willful slave. If this mechanism would not have been implemented, there could be potential inefficiency when a device that happened to be synchronized to another Master loses arbitration, but is not able to respond to the winning device.

Another situation for arbitration loss could be a bus exception resulting from a device operating not according to the bus protocol or

Using the 8XC751/752 in multimaster I²C applications AN430

interference on the bus lines. In addition to "regular" arbitration loss detected with the ARL hardware flag, such a situation may occur with detecting a Start or a Stop in the middle of transmitting an address or data byte. In such a situation the microcontroller withdraws from the bus as well—active Slave upon a Start detection, or returning as an idle slave in other cases.

When a Master transaction is terminated by an arbitration loss, the Master Request flag (MASTRQ) of the hardware I²C port remains in effect. As a result when the bus gets free, our device will take control, issue a Start, and the transaction that was cut will start again. This restart will happen automatically, without any application involvement (unlike non-acknowledgement, where the MastNext routine determines what shall be done).

The I²C communications routines are structured as an interrupt service routine responding to an I²C port interrupt upon a frame Start. Within a frame the I²C processing is continuous, where the I²C port is polled for hardware response, and the I²C interrupts are disabled. Other interrupts are enabled during the service routine. The set-up requirements from the mainline program are minimal, and interfacing is done via RAM buffers and some pre defined RAM locations. The lower level interface with the hardware is done inside the service routine, and can typically be ignored by the application programmer.

BUS WATCHDOG AND ERROR RECOVERY

A malfunctioning device (in hardware or software) may hold the SCL line low, thus causing the bus to be "stuck". It might even be possible that a transient protocol violation (due to hardware interference, such as a device turning on) may cause some devices (non programmable, or even microcontrollers which were not carefully programmed) to hold the bus. Since within a frame the bus is software-pollled, a "stuck" bus might cause the application software to "hang forever". Here the TIMER1 watchdog comes to the rescue, interrupting when there is no bus activity for a long period of time.

When the I²C service routine is interrupted by the watchdog timer, the processing of the current frame is not completed and the event routines are not called. The software returns to execute the mainline application, and will be interrupted again for the next frame (next Start, received as a slave or induced as a Master). A status flag and a counter report on the watchdog interrupt, so the application program can be made to inhibit the I²C port if

there are too many occurrences of a "hanging" bus.

Bus protocol errors and "hangups" might be an issue in systems which are susceptible to noise, temporary bus line shorts, "hot plug in" of devices or even erroneously programmed devices—and a "fail safe" controller program should be able to detect bus problems and possibly assist in resolving them. The RECOVER routine resets the I²C interface of the microcontroller, and attempts to release some other devices on the bus by toggling the clock line. The I²C interface of the 8XC751 is reset by letting Timer1 run and expire, since this circuitry does not feature a software controlled reset. This "extreme" measure is needed in some cases of bus protocol violation.

The bus and interface circuit recovery routine can be automatically invoked whenever Timer1 detects a timeout. In addition, for systems where potential bus failures are a concern and reliability is an issue, one may implement mechanisms to invoke bus and interface recovery from the application code. This may help in cases where the bus gets "stuck" when there is no I²C frame in progress. In such an instance the watchdog timer will not give any timeout indications, as it has not been activated. Another case emanates from a design peculiarity of the interface circuitry on the 8XC751: if the SCL line is externally grounded when there is a Start condition, this Start might be ignored, and the watchdog may not be activated. Our programming example deals with potential failures by testing for transaction completion and retrying transmissions when necessary (these are explicit retries, in addition to an "automatic" retry after a Master's arbitration loss, invoked by the MASTRQ bit). Too many transmission failures activate the RECOVER routine.

I²C COMMUNICATIONS ROUTINES—INTERFACE

The I²C service routine deals with the transmission and reception of messages, without any concern for the contents of the message. In order to provide a general interface for different applications the data is transferred via buffers. The service routine does not have to "know" where the data goes to or comes from—as long as the application program specifies the required pointers for these buffers. The interface to the actual application (which "cares" about message contents, timing, addressing and so forth) is done in a well defined manner, allowing usage of the same service routine with different application programs.

The interface is carried out with the use of buffers, pre-defined names for Application Event Routines, interface RAM locations for transferring parameters, pointers and flags, and constants. A more detailed discussion of the interface follows.

Buffers

There are three buffers for data transfers between the I²C bus and the application program.

MasBuf is used for Master transmission and reception. The number of data bytes for each Master message—reception or transmission, is specified by the memory location MASTCNT. The value in MASTCNT should be less than the length of MasBuf. For Master transmission the message is placed in MasBuf before the transmission is initiated. In Master reception, the received message will be contained in the same buffer. There is only one Master message transaction occurring at the same time, so we may use the same buffer both for transmission and reception.

For Slave operation we must accommodate data transfers which may come randomly, asynchronous to each other or to possible operation of the same device as a Master. Therefore it is necessary to allocate additional RAM area as buffers dedicated to Slave operation: SRcvBuf for receiving data, STxBuf for transmission.

The length of the Slave receive buffer is defined by the symbol RBufLen. It is used by the code for protection, avoiding overwriting RAM beyond the allocated buffer size in case a Master sends a message which is too long. There is no need for RAM protection for transmission, but the Master should not request more data than STxBuf can supply.

Interface RAM Locations

RAM location MyAddr contains the address of this processor.

Status flag MSGSTAT is used for reporting to the application on I²C communications status—mainly on the successful, or unsuccessful, completion of a message transaction. The contents of MSGSTAT may be used by the mainline application code or by the Event Routines. The different codes that could be placed by the I²C service routine are described later in the text. When the message processing commences, a code indicating Slave or Master processing is inserted to MSGSTAT, and is updated as we go along. There could be many applications that will not need to use MSGSTAT contents, as the very fact of calling a certain event routine implies completion of a processing stage.

Using the 8XC751/752 in multimaster I²C applications AN430

For Master transactions, in addition to the data buffer MasBuf, there are several RAM locations into which the application inserts Master message "directives". These directives provide the service routine with the information necessary to carry out the next Master transaction. The one byte RAM locations used for directives are DESTADRW, DESSUBAD, MASTCNT and MASCMD.

DESTADRW contains the destination slave address in bits 7-1, while bit 0 is the R/W bit. Bit 0 contains 0 for a Write operation (the message is to be transmitted to the slave) and 1 for a Read operation (message is being read from the slave and received by this Master).

DESSUBAD contains the 8 bit sub-address of the slave, if necessary. For transactions without a sub-address, the contents of DESSUBAD is ignored.

MASTCNT contains the number of data bytes in the message to be sent from or received into MasBuf. This number should not be bigger than the length of MasBuf.

MASCMD byte contains the bit flags SUBADD, RPSTRT and SETMRQ. SUBADD is 0 (cleared) for a message with a regular address, and 1 (set) when a subaddress is required. When SUBADD is set, the service routine takes care of all the protocol required for sub-addressing, which includes a Repeated Start for Read operations. A message with a subaddress is considered to be a single message, even if it includes a Repeated Start.

The RPSTRT and SETMRQ are kept cleared in regular applications, and will be used only for "tailoring" the bus transfers in special cases. When RPSTRT is cleared the message will terminate, as usually required, with a Stop. When RPSTRT is set a Repeated Start will be sent on the bus, and Master operation will resume. The RPSTRT directive relates to terminating the message after all the data was transferred, and not to the mandatory Repeated Start in the middle of sub-addressed Read operation. A single message with a subaddress will typically have RPSTRT cleared. SETMRQ indicates what will be loaded into the MASTRQ flag of the hardware when Stop is transmitted. Typically it will be cleared. When SETMRQ is 1, MASTRQ will be set, thus trying to issue a new Start immediately following the Stop. In such a case the service routine will not return upon Stop, but will continue as a Master.

TITOCNT is used to count time-outs of the watchdog timer. Whenever such a timeout invokes the TIMER1 interrupt service routine the contents of the location TITOCNT are

incremented, and the timeout is reported in MSGSTAT. The count is saturated at 0FFh. This mechanism may be used in an application that is very much "concerned" with potential bus failures, allowing some type of "failure monitoring" by the application even for Slave transactions.

APPLICATION EVENT ROUTINES

The service routine calls Event Routines with pre-defined names (Figure 16), and these routines must be provided by the application program. The actual code of the routines will differ from application to application, but the routine names are being kept the same.

These routines are being called when successful processing of a message (send or receive) is completed. The routines may perform whatever action the application was designed for, which is not necessarily related to the I²C communications mechanism. In addition, the routines may perform the data interface tasks for the I²C port, like emptying buffers from received data or preparing the next message by setting up the buffers.

The mechanism of calling the event routines out of the service routine allows an immediate reaction to the event of message processing completion, before any new activity happens on the bus. In some simple applications this may not be necessary. For example, one may have a main program for a slave which is just a wait loop monitoring a flag set by the service routine when a message transfer, initiated by some master, is completed. In such a case the application could react to the message completion after the interrupt service routine returns. However, in the general case this will not be sufficient. An example could be a slave with an application which is constantly busy doing another task, in an environment where the communication requests on the I²C bus are frequent. If there is a new message request shortly after the current message is completed, having to wait for the application until it "has time" may result in not reacting, or sending the same data again, or overwriting the received data in the buffer. Another obvious case demanding event routine calls is a Master sending different messages with a Repeated Start—the new data for the following message must be prepared in the interrupt service routine as the current message is completed (there is no return from interrupt prior to the new data transmission).

The programmer has the flexibility to decide where to prepare the next message according to the requirements of the

application. This can be done after return from the event routine, in the application code after the return from interrupt, or a combination of both, where the time critical events are performed in the event routines. The application may monitor the MSGSTAT flag for message processing completion. If the event routines are not used, it is recommended to simply code them as a "RET" instruction, thus turning them into dummy routines (this is an easier and better practice than changing the service routine itself, eliminating the calls).

Master Event Routine:

MastNext

This routine is called by the service routine when the processing of the current Master message is completed. For an indication on the type of message processing completion, MastNext may inspect the contents of MSGSTAT RAM location.

When MastNext is called, MSGSTAT will contain one of the following codes for message processing completion:

MRCVED (= 21h)—a complete message (with number of data bytes indicated by MASTCNT) was received from the slave.

MTXED (= 22h)—the number of data bytes indicated by MASTCNT were successfully sent and acknowledged by the slave.

MTXNAK (= 23h)—the slave did not acknowledge a data byte of the message, even though it had acknowledged its address. The message transmission was terminated upon the NAK.

MTXNOSLV (= 24h)—no slave acknowledged the address indicated by memory location DESTADDR.

The MastNext routine may perform any task(s) necessary for the application. Data handling tasks will typically be dependent on the MSGSTAT indication. One possible task could be setting the directives for the next message. The necessity for executing this task here (versus the main-line code initiating the transfer) is of course application dependent.

Slave Event Routines:

These routines are called when a message transaction as a slave has been completed. In many cases it could be important to utilize the calls to such routines as the requests for message transactions as a slave can come randomly, asynchronous to the application program. The application may demand that new data coming in should immediately

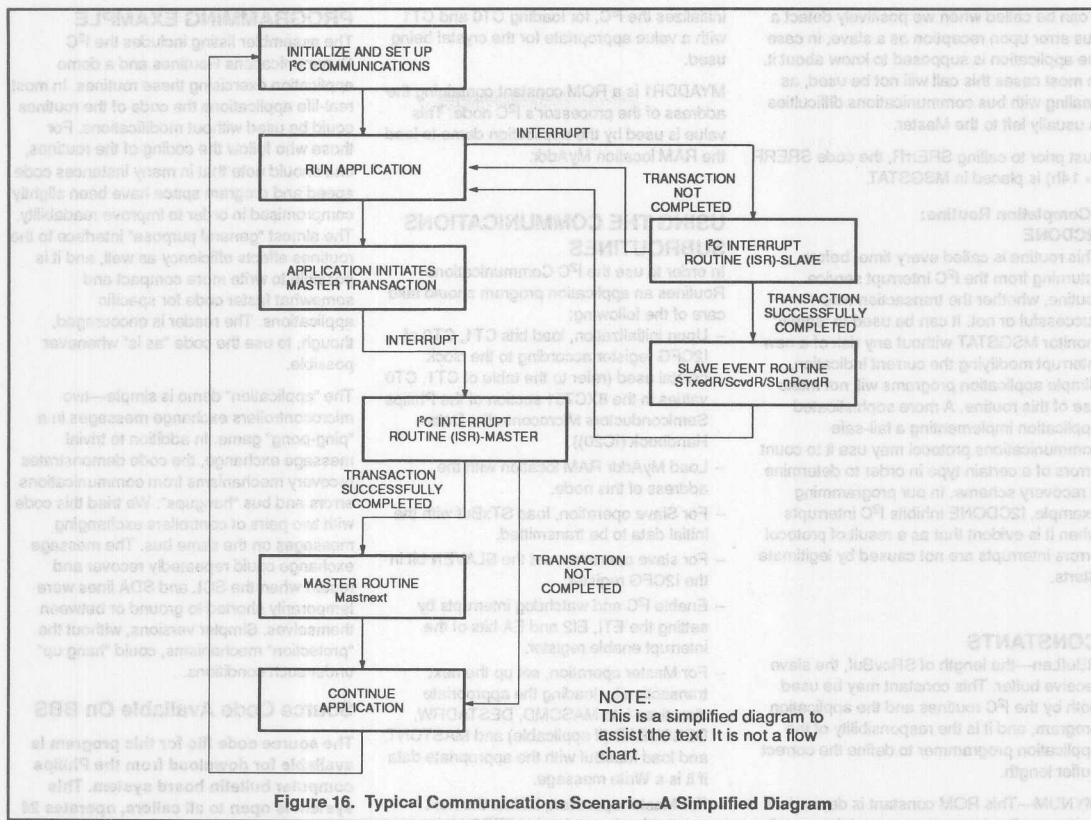
Using the 8XC751/752 in multimaster I²C applications AN430

Figure 16. Typical Communications Scenario—A Simplified Diagram

initiate some tasks (e.g. control an output port)—and the event routine can be used to process the result of the slave interrupt.

In most cases it will be necessary for a slave to react immediately to a message received simply in order not to lose the data. As a new message may come randomly, it may overwrite the reception buffer before the data has been transferred out of it or acted upon.

For applications in which the reaction for slave events is performed after the return from the service routine, the event is reported by placing an appropriate code in the MSGSTAT flag. The programmer may use event routines, other mainline routines inspecting MSGSTAT, or both. If the event routines are not used, it is recommended to code them as a "RET" instruction.

SRcvdR:

Called by the service routine when a new, complete message has been received into SRcvBuf. When SRcvdR is called, R1 points

to the address of the last byte received into the buffer. In a typical application SRcvdR will transfer the new data out of SRcvBuf, so it will not be written over by a subsequent slave reception.

The equivalent MSGSTAT indication for this event is SRCVD (= 11h).

SLnRcvdR:

Called when a slave message has been received into SRcvBuf, but the message was longer than the SRcvBuf buffer (as specified by RbufLen).

The equivalent MSGSTAT indication for this event is SRLNG (= 12h).

If the program is supposed to react to a too long a message the same way as to a message that can be contained in the buffer, one may code SLnRcvdR simply as a call to SRcvdR.

STXedR:

Called by the service routine when data has been transmitted out of the slave STxBuf buffer according to a master's request. This routine may insert new data into the buffer, preparing it for the next slave transmission.

The equivalent MSGSTAT indication for this event is STXED (= 13h).

Note that we do not have a separate routine for the case that the master requested too many bytes—more than STxBuf length—and we sent out meaningless bytes. It is the master's responsibility to specify the message length, and it should be able to request messages with the appropriate length from each slave on the bus.

SRErrR:

This routine relates more to bus communications than to the application itself.

Using the 8XC751/752 in multimaster I²C applications AN430

It can be called when we positively detect a bus error upon reception as a slave, in case the application is supposed to know about it. In most cases this call will not be used, as dealing with bus communications difficulties is usually left to the Master.

Just prior to calling `SRERRR`, the code `SRERR` (= 14h) is placed in `MSGSTAT`.

0Completion Routine: I2CDONE

This routine is called every time, before returning from the I²C interrupt service routine, whether the transaction was successful or not. It can be used to "safely" monitor `MSGSTAT` without any risk of a new interrupt modifying the current indication. Simple application programs will not make use of this routine. A more sophisticated application implementing a fail-safe communications protocol may use it to count errors of a certain type in order to determine a recovery scheme. In our programming example, `I2CDONE` inhibits I²C interrupts when it is evident that as a result of protocol errors interrupts are not caused by legitimate Starts.

CONSTANTS

`RBuLen`—the length of `SRcvBuf`, the slave receive buffer. This constant may be used both by the I²C routines and the application program, and it is the responsibility of the application programmer to define the correct buffer length.

`MYNUM`—This ROM constant is dependent on the application environment. It is a small integer defining a "serial number" of the node, out of all the processors running the same code. This constant is used only when recovering from a timeout, in order to "de-synchronize" masters from each other when trying to recover the bus.

`CTVAL1` is a constant defined in ROM. It is used by the application code portion which

initializes the I²C, for loading `CT0` and `CT1` with a value appropriate for the crystal being used.

`MYADDR1` is a ROM constant containing the address of the processor's I²C node. This value is used by the application demo to load the RAM location `MyAddr`.

USING THE COMMUNICATIONS SUBROUTINES

In order to use the I²C Communications Routines an application program should take care of the following:

- Upon initialization, load bits `CT1`, `CT0` of `I2CFG` register according to the clock crystal used (refer to the table of `CT1`, `CT0` values in the 8XC751 section of the Philips Semiconductors Microcontroller Data Handbook (IC20)).
- Load `MyAddr` RAM location with the address of this node.
- For Slave operation, load `STxBuf` with the initial data to be transmitted.
- For slave operation, set the `SLAVEN` bit in the `I2CFG` register.
- Enable I²C and watchdog interrupts by setting the `ETI`, `EI2` and `EA` bits of the interrupt enable register.
- For Master operation, set up the next transaction by loading the appropriate directives into `MASCMD`, `DESTADRW`, `DESSUBAD` (if applicable) and `MASTCNT`, and load `MasBuf` with the appropriate data if it is a Write message.
- For Master operation, initiate the next transaction by setting `MASTRQ` bit in `I2CFG`.
- For both Master and Slave operation, handle data transmission and reception via the buffers in main-line code or the Event Routines.

PROGRAMMING EXAMPLE

The assembler listing includes the I²C Communications Routines and a demo application exercising these routines. In most real-life applications the code of the routines could be used without modifications. For those who follow the coding of the routines, one should note that in many instances code speed and program space have been slightly compromised in order to improve readability. The almost "general purpose" interface to the routines affects efficiency as well, and it is possible to write more compact and somewhat faster code for specific applications. The reader is encouraged, though, to use the code "as is" whenever possible.

The "application" demo is simple—two microcontrollers exchange messages in a "ping-pong" game. In addition to trivial message exchange, the code demonstrates recovery mechanisms from communications errors and bus "hangups". We tried this code with two pairs of controllers exchanging messages on the same bus. The message exchange could repeatedly recover and restart when the `SCL` and `SDA` lines were temporarily shorted to ground or between themselves. Simpler versions, without the "protection" mechanisms, could "hang up" under such conditions.

Source Code Available On BBS

The source code file for this program is available for download from the Philips computer bulletin board system. This system is open to all callers, operates 24 hours a day, and can be accessed with modems at 2400, 1200, and 300 baud. The telephone numbers for the BBS are: (800) 451-6644 (in the U.S. only) or (408) 991-2406.

Using the 8XC751/752 in multimaster I²C applications AN430

PPCODE1 83C751 Multimaster I2C Routines

4/14/1992 PAGE 1

```

1      ;
2
3      ;*****
4      ;           Multimaster Code for 83C751/83C752
5      ;           4/14/1992
6      ;*****
7      ; This code was written to accompany an application note. The I2C routines
8      ; are intended to be demonstrative and transportable into different
9      ; application scenarios, and were NOT optimized for speed and/or memory
10     ; utilization.
11     ;
12     ; Yoram Arbel
13
14     $TITLE(83C751 Multi Master I2C Routines)
15     $DATE(4/14/1992)
16     $MOD751
17     $DEBUG
18
19     ;*****
20     ;           8XC751 MULTIMASTER I2C COMMUNICATIONS ROUTINES
21     ;           Symbols and RAM definitions
22     ;*****
23
24     ; Symbols (masks) for I2CFG bits.
25
0010 26     BTIR      EQU      10h      ; TIRUN bit.
0040 27     BMRQ      EQU      40h      ; MASTRQ bit.
28
29
30     ; Symbols (masks) for I2CON bits.
31
0080 32     BCXA      EQU      80h      ; CXA bit.
0040 33     BIDL      EQU      40h      ; IDLE bit.
0020 34     BCDR      EQU      20h      ; CDR bit.
0010 35     BCARL     EQU      10h      ; CARL bit.
0008 36     BCSTR     EQU      08h      ; CSTR bit.
0004 37     BCSTP     EQU      04h      ; CSTP bit.
0002 38     BXSTR     EQU      02h      ; XSTR bit.
0001 39     BXSTP     EQU      01h      ; XSTP bit.
40
41     ; Note:
42     ;
43     ; Specific bits of the I2CON register are set by writing into this register a
44     ; combination of the masks defined above using the MOV command.
45     ; The SETB command should not be used with I2CON, as it is implemented by
46     ; reading the contents of the register, setting the appropriate bit and
47     ; writing it back into the register. As the functionality of the Read and
48     ; Write portions of the I2CON register is different, using SETB may cause
49     ; unwanted results.
50
51     ; Message transaction status indications in MSGSTAT:
52

```

Using the 8XC751/752 in multimaster I²C applications AN430

PPCODE1 83C751 Multimaster I2C Routines

4/14/1992 PAGE 2

0010	53	SGO	EQU	10h	; Started Slave message processing.
0011	54	SRCVD	EQU	11h	; as a slave, received a new message
0012	55	SRLNG	EQU	12h	; received as slave a message which is too
	56				; long for the buffer
0013	57	STXED	EQU	13h	; as slave, completed message transmission.
0014	58	SRERR	EQU	14h	; bus error detected when operating as a slave.
	59				
0020	60	MGO	EQU	20h	; Started Master message processing.
0021	61	MRCVED	EQU	21h	; As Master, received complete message from
	62				; slave.
0022	63	MTXED	EQU	22h	; As Master, completed successful message
	64				; transmission (slave acknowledged all data
	65				; bytes).
0023	66	MTXNAK	EQU	23h	; As Master, truncated message since slave did
	67				; not acknowledge a data byte.
0024	68	MTXNOSLV	EQU	24h	; AS Master, did not receive an acknowledgement
	69				; for the specified slave address.
	70				
0030	71	TIMOUT	EQU	30h	; TIMER1 Timed out.
0032	72	NOTSTR	EQU	32h	; Master did not recognize Start.
	73				
	74				; RAM locations used by I2C interrupt service routines.
	75				
	76				
0020	77	MASCMD	DATA	20h	
0000	78	SUBADD	BIT	MASCMD.0	
0001	79	RPSTRT	BIT	MASCMD.1	
0002	80	SETMRQ	BIT	MASCMD.2	
	81				
0024	82	DSEG	AT	24h	
	83				
0024	84	MSGSTAT:	DS	1	; I2C communications status.
0025	85	MYADDR:	DS	1	; Address of this I2C node.
0026	86	DESTADRW:	DS	1	; Destination address + R/W
					(for Master).
0027	87	DESSUBAD:	DS	1	; Destination subaddress.
0028	88	MASTCNT:	DS	1	; Number of data bytes in message (Master,
	89				; send or receive).
	90				
0029	91	TITOCNT:	DS	1	; Timer I bus watchdog timeouts counter.
002A	92	StackSave:	DS	1	; SP save location (used when returning from
	93				; bus recovery routine).
	94				
002B	95	MasBuf:	DS	4	; Master receive/transmit buffer, 8 bytes.
002F	96	SRcvBuf:	DS	4	; Slave receive buffer, 8 bytes.
0033	97	STxBuf:	DS	4	; Slave transmit buffer, 8 bytes.
	98				
	99				
	100				
0004	101	RBufLen	EQU	4h	; The length of SRcvBuf
	102				

Using the 8XC751/752 in multimaster I²C applications AN430

PPCODE1 83C751 Multimaster I2C Routines

4/14/1992 PAGE 3

```

103 ;*****
104 ; APPLICATION output pins and RAM definitions
105 ;*****
106
107 ; Outputs used by the application:
108
0090 109 TogLED BIT P1.0 ; Toggling output pin, to confirm
110 ; that the ping-pong game proceeds fine.
0091 111 ErrLED BIT P1.1 ; Error indication.
112
0093 113 OnLED BIT P1.3
114
115 ; Application RAM
116
0021 117 APPFLAGS DATA 21h
0008 118 TRQFLAG BIT APPFLAGS.0
119 ; Flag for monitoring I2C transmission success.
0009 120 SErrFLAG BIT APPFLAGS.1
121
0037 122 FAILCNT: DS 1
123
0038 124 TOGCNT: DS 1 ; Toggle counter.
125
126
127 ;*****
128 ;
129 ; Program Start
130 ;
131 ;*****
132 CSEG
133
134 ; Reset and interrupt vectors.
135
0000 4178 136 AJMP Reset ;Reset vector at address 0.
137
138
139 ; A timer I timeout usually indicates a 'hung' bus.
140
001B 141 ORG 1Bh ; Timer I (I2C timeout) interrupt.
001B D2DD 142 TimerI: SETB CLRTI
001D 4111 143 AJMP TIISR ; Go to Interrupt Service Routine.
144
145 ;*****
146
147
148 ;*****
149 ; I2C Interrupt Service Routine
150 ;*****
151 ;
152 ; Notes on the interrupt mechanism:
153 ;
154 ; Other interrupts are enabled during this ISR upon return from XRETI.

```

Using the 8XC751/752 in multimaster I²C applications AN430

PPCODE1 83C751 Multimaster I2C Routines

4/14/1992

PAGE 4

```

155 ; Limitations imposed on other ISR's:
156 ; - Should not be long (close to 1000 clock cycles). A long ISR will cause
157 ; the I2C bus to "hang", and a TIMERI interrupt to occur.
158 ; - Other interrupts either do not use the same mechanism for allowing
159 ; further interrupts, or if they do - disable TIMERI interrupt beforehand.
160 ;
161 ; The 751 hardware allows only one level of interrupts. We simulate an
162 ; additional level by software: by performing a RETI instruction (at location
163 ; XRETI) the interrupt-in-progress flip-flop is cleared, and other interrupts
164 ; are enabled. The second level of interrupt is a must in our implementation,
165 ; enabling timeout interrupts to occur during "stuck" wait loops in the I2C
166 ; interrupt service routine.
167
168
0023 169 ORG 23h
170
0023 C2AC 171 I2CISR: CLR EI2 ; Disable I2C interrupt.
0025 114C 172 ACALL XRETI ; Allow other interrupts to occur.
0027 C0D0 173 PUSH PSW
0029 C0E0 174 PUSH ACC
002B E8 175 MOV A,R0
002C C0E0 176 PUSH ACC
002E E9 177 MOV A,R1
002F C0E0 178 PUSH ACC
0031 EA 179 MOV A,R2
0032 C0E0 180 PUSH ACC
181
0034 85812A 182 MOV StackSave, SP
0037 C2DC 183 CLR TIRUN
0039 D2DC 184 SETB TIRUN
185
003B 209A09 186 JB STP,NoGo
003E 30990C 187 JNB MASTER, GoSlave
0041 752420 188 MOV MSGSTAT,#MGO
0044 209B76 189 JB STR,GoMaster
0047 752432 190 NoGo: MOV MSGSTAT,#NOTSTR
004A 21AE 191 AJMP Dismiss ; Not a valid Start.
192
004C 32 193 XRETI: RETI
194
195 ;*****
196 ; Main Transmit and Receive Routines
197 ;*****
198
199 ; SLAVE CODE -
200 ; GET THE ADDRESS
201
004D 752410 202 GoSlave: MOV MSGSTAT,#SGO
0050 31E2 203 AddrRcv: ACALL CIsRcv8
0052 309D5E 204 JNB DRDY, SMsEnd ; Must be some strange Start or Stop
205 ; before the address byte was completed.
206 ; Not a valid address.

```


Using the 8XC751/752 in multimaster I²C applications AN430

PPCODE1 83C751 Multimaster I2C Routines

4/14/1992 PAGE 5

0055 A2E0	207	STstRW:	MOV C,ACC.0 ; Save R/W~ bit in carry.	
0057 C2E0	208	CLR	ACC.0 ; Clear that bit, leaving "raw" address	
0059 6060	209	JZ	GoIdle ; If it is a General Address	
	210		; - ignore it.	
	211			
	212		; NOTE:	
	213		; One may insert here a different	
	214		; treatment for general calls, if	
	215		; these are relevant.	
	216			
005B 4027	217	JC	SlvTx ; It's a Read - (requesting slave	
	218		; transmit).	
	219			
	220			
	221			
	222			
	223		; It is a Write (slave should receive the message).	
	224			
	225		; Check if message is for us	
	226			
005D B5255B	227	SRcv2:	CJNE A,MYADDR,GoIdle ; If not my address - ignore the	
	228		; message.	
0060 792F	229	MOV	R1,#SRcvBuf ; Set receive buffer address.	
0062 7A05	230	MOV	R2,#RbufLen+1 ;	
0064 8002	231	SJMP	SRcv3	
	232			
0066 F7	233	SRcvSto:	MOV @R1,A ; Store the byte	
0067 09	234	Inc	R1 ; Step address.	
0068 31ED	235	SRcv3:	ACALL AckRcv8	
006A 309D09	236	JNB	DRDY,SRcvEnd ; Exit loop - end reception.	
006D DAF7	237	DJNZ	R2,SRcvSto ; Go to store byte if buffer not full.	
	238			
	239		; Too many bytes received - do not acknowledge.	
006F 752412	240	MOV	MSGSTAT,#SRLNG ; Notify main that (as slave) we	
	241		; have received too long a message.	
0072 7110	242	ACALL	SLnRCvdR ; Handle new data - slave event routine.	
0074 8045	243	SJMP	GoIdle	
	244			
	245			
	246			
	247		; Received a byte, but not DRDY - check if a legitimate message end.	
	248			
0076 B8072E	249	SRcvEnd:	CJNE R0,#7,SRcvErr ; If bit count not 7, it was not	
	250		; a Start or a Stop.	
	251			
	252		; Received a complete message	
	253			
	254			
0079 752411	255	MOV	MSGSTAT,#SRCVD	
	256		; Calculate number of bytes received	
007C E9	257	MOV	A,R1	
007D C3	258	CLR	C	

Using the 8XC751/752 in multimaster I²C applications AN430

PPCODE1 83C751 Multimaster I2C Routines

4/14/1992 PAGE 6

007E 942F	259	SUBB	A,#SRcvBuf	; number of bytes in ACC
0080 51EF	260	ACALL	SRcvdR	; Handle new data – slave event routine.
0082 802F	261	SJMP	SMsgEnd	
	262			
	263			
	264			; It is a Read message, check if for us.
	265			
0084 00	266	SlvTx:	NOP	
	267			
0085 B52533	268	STx2:	CJNE A,MYADDR,GoIdle	; Not for us.
0088 759900	269	MOV	I2DAT,#0	; Acknowledge the address.
008B 309EF0	270	JNB	ATN,\$; Wait for attention flag.
008E 309D22	271	JNB	DRDY,SMsgEnd	; Exception – unexpected Start
	272			; or Stop before the Ack got out.
0091 7933	273	MOV	R1,#STxBuf	; Start address of transmit buffer.
0093 E7	274	STxlp:	MOV A,@R1	; Get byte from buffer
0094 09	275	INC	R1	
0095 31CE	276	ACALL	XmByte	
0097 309D19	277	JNB	DRDY,SMsgEnd	; Byte Tx not completed.
009A 309FF6	278	JNB	RDAT,STxlp	; Byte acknowledge, proceed trans.
009D 759860	279	MOV	I2CON,#BCDR+BDLE	; Master Nak'ed for msg end.
00A0 752413	280	MOV	MSGSTAT,#STXED	
00A3 7110	281	ACALL	STXedR	; Slave transmitted event routine.
00A5 21AE	282	AJMP	Dismiss	
	283			
	284			
00A7 752414	285	SRcvErr:	MOV MSGSTAT,#SRERR	; Flag bus/protocol error
00AA 7110	286	ACALL	SRErrR	; Slave error event routine.
00AC 8005	287	SJMP	SMsgEnd	
00AE 752414	288	StxErr:	MOV MSGSTAT,#SRERR	; Flag bus/protocol error
00B1 7110	289	ACALL	SRErrR	
	290			
00B3 209903	291	SMsgEnd:	JB MASTER,SMsgEnd2	
00B6 209B94	292	JB	STR,GoSlave	; If it was a Start, be Slave
00B9	293	SMsgEnd2:		
00B9 21AE	294	AJMP	Dismiss	
	295			
	296			
	297			; End of Slave message processing
	298			
00BB	299	GoIdle:		
00BB 21AE	300	AJMP	Dismiss	
	301			
	302			
	303			
	304			
	305			
	306			
	307			
00BD	308	GoMaster:		
	309			
	310			

Using the 8XC751/752 in multimaster I²C applications AN430

PPCODE1	83C751 Multimaster I2C Routines	4/14/1992	PAGE 7
	311 ; Send address & R/W~ byte		
	312		
00BD 792B	313 MOV R1,#MasBuf ; Master buffer address		
00BF AA28	314 MOV R2,MASTCNT ; # of bytes, to send or rcv		
00C1 E526	315 MOV A,DESTADRW ; Destination address (including		
	316 ; R/W~ byte).		
00C3 200012	317 JB SUBADD,GoMas2 ; Branch if subaddress is needed.		
	318		
00C6 31C5	319 ACALL XmAddr		
	320		
00C8 309D03	321 JNB DRDY,GM2		
00CB 309C02	322 JNB ARL,GM3		
00CE 2186	323 GM2: AJMP AdTxAr1 ; Arbitration loss while transmitting		
	324 ; the address.		
00D0 209F5C	325 GM3: JB RDAT,Noslave ; No Ack for address transmission.		
00D3 20E063	326 JB ACC.0, MRcv ; Check R/W~ bit		
00D6 211A	327 AJMP MTx		
	328		
	329 ; Handling subaddress case:		
	330		
00D8 00	331 GoMas2: NOP ; Subaddress needed. Address in ACC.		
00D9 C2E0	332 CLR ACC.0 ; Force a Write bit with address.		
00DB 31C5	333 ACALL XmAddr		
00DD 309D03	334 JNB DRDY,GM4		
00E0 309C02	335 JNB ARL,GM5		
00E3 2186	336 GM4: AJMP AdTxAr1 ; Arbitration loss while transmitting		
	337 ; the address.		
	338		
00E5 209F47	339 GM5: JB RDAT,Noslave ; No Ack for address transmission.		
00E8 E527	340 MOV A,DESSUBAD		
00EA 31CE	341 ACALL XmByte ; Transmit subaddress.		
00EC 309DCA	342 JNB DRDY,SMsEnd2 ; Arbitration loss (by Start or Stop)		
00EF 209CC7	343 JB ARL,SMsEnd2 ; Arbitration loss occurred.		
00F2 209F3F	344 JB RDAT,NoAck ; Subaddress transmission was not ack'ed.		
00F5 E526	345 MOV A,DESTADRW ; Reload ACC with address.		
00F7 30E020	346 JNB ACC.0, MTx ; It's a Write, so proceed		
	347 ; by sending the data.		
	348 ; Read message, needs rp. Start and add. retransmit.		
	349		
00FA 759822	350 MOV I2CON,#BCDR+BXSTR ; Send Repeated Start.		
00FD 309EFD	351 JNB ATN,\$		
0100 759820	352 MOV I2CON,#BCDR ; Clear useless DRDY while preparing		
	353 ; for Repeated Start.		
0103 309EFD	354 JNB ATN,\$; expecting an STR.		
0106 309C02	355 JNB ARL,GM6		
0109 2182	356 AJMP MArlEnd ; oops - lost arbitration.		
010B 31C5	357 GM6: ACALL XmAddr ; Retransmit address, this time with the		
	358 ; Read bit set.		
010D 309D03	359 JNB DRDY,GM7		
0110 309C02	360 JNB ARL,GM8		
0113 2186	361 GM7: AJMP AdTxAr1 ; Arbitration loss while transmitting		
	362 ; the address.		

Using the 8XC751/752 in multimaster I²C applications AN430

PPCODE1 83C751 Multimaster I2C Routines

4/14/1992 PAGE 8

```

0115 209F17 363 GM8:      JB      RDAT,Noslave ; No Ack – the slave disappeared.
0118 801F    364 SJMP     MRcv      ; Proceed receiving slave's data.
                                365
                                366 ; A Write message. Master transmits the data.
                                367
011A 00      368 MTx:      NOP
                                369
011B E7      370 MTxLoop: MOV     A,@R1 ; Get byte from buffer.
011C 09      371 INC       R1          ; Step the address.
011D 31CE    372 ACALL    XmByte
011F 309D97 373 JNB      DRDY,SMsgEnd2 ; Arbitration loss (by Start or Stop)
0122 209C94 374 JB      ARL,SMsgEnd2 ; Arbitration loss.
0125 209F0C 375 JB      RDAT,NoAck
0128 DAF1    376 DJNZ     R2,MTxLoop ; Loop if more bytes to send.
                                377
012A 752422 378 MOV      MSGSTAT,#MTXED; Report completion of buffer
                                379 ; transmission.
012D 8025    380 SJMP     MTxStop
012F 752424 381 NoSlave: MOV     MSGSTAT,#MTXNOSLV
0132 8020    382 SJMP     MTxStop
0134 752423 383 NoAck:   MOV     MSGSTAT,#MTXNAK
0137 801B    384 SJMP     MTxStop
                                385
                                386
                                387
                                388 ; Master receive – a Read frame
                                389
0139 31F6    390 MRcv:    ACALL   ClaRcv8 ; Receive a byte.
013B 8002    391 SJMP     MRcv2
013D 31ED    392 MRcvLoop: ACALL  AckRcv8
013F 309D39 393 MRcv2:   JNB     DRDY,Marl ; Other's Start or Stop.
0142 F7      394 MOV      @R1,A ; Store received byte.
0143 09      395 INC      R1 ; Advance address.
0144 DAF7    396 DJNZ     R2,MRcvLoop
                                397
                                398 ; Received the desired number of bytes – send Nack.
                                399
0146 759980 400 MOV      I2DAT,#80h ;
0149 309EFD 401 JNB      ATN,$
014C 309D2C 402 JNB      DRDY,Marl
014F 752421 403 MOV      MSGSTAT,#MRCVED
0152 8000    404 SJMP     MTxStop ; Go to send Stop or Repeated Start.
                                405
                                406
                                407
                                408 ; Conclude this Master message:
                                409 ; Send Stop, or a Repeated Start
                                410
                                411
0154 300105 412 MTxStop: JNB     RPSTRT,MTxStop2 ; Check if Repeated Start needed
                                413 ; Around if not RPSTRT.
0157 759822 414 MOV      I2CON,#BCDR+BXSTR ; Send Repeated Start.

```

Using the 8XC751/752 in multimaster I²C applications AN430

PPCODE1 83C751 Multimaster I2C Routines

4/14/1992

PAGE 9

```

015A 8007    415    SJMP      MTxStop3
015C A202    416    MTxStop2: MOV      C,SETMRQ ; Set new Master Request if demanded
015E 92DE    417    MOV      MASTRQ,C ; by SETMRQ bit of MASCMD.
                                418
0160 759821  419    MOV      I2CON,#BCDR+BXSTP ; Request the HW to send a Stop.
                                420
0163 309EFD  421    MTxStop3: JNB      ATN,$ ; Wait for Attention
0166 759820  422    MOV      I2CON,#BCDR ; Clear the useless DRDY, generated
                                423 ; by SCL going high in preparation
                                424 ; for thr Stop or Repeated Start.
0169 309EFD  425    JNB      ATN,$ ; Wait for ARL, STP or STR.
016C 209C13  426    JB       ARL,MarlEnd ; Lost arbitration trying to send
                                427 ; Stop or a ReStart.
                                428
                                429 ; Master is done with this message. May proceed with new messages, if any,
                                430 ; or exit.
                                431
016F 7112    432    ACALL     MastNext ; Master Event Routine. May Prepare
                                433 ; the pointers and data for the
                                434 ; next Master message.
                                435
0171 30DE05  436    JNB      MASTRQ,MMsEnd; Go end service routine if MASTRQ
                                437 ; does not indicate that the master
                                438 ; should continue (was set according
                                439 ; to SETMRQ bit, or by MastNext).
                                440
0174 309B02  441    JNB      STR,MMsEnd ; Return from the ISR, unless Start
                                442 ; (avoid danger if we do not return:
                                443 ; if there was a Stop, the watchdog
                                444 ; is inactive until next Start).
0177 01BD    445    AJMP     GoMaster ; Loop for another Master message
                                446 ;
0179         447    MMsEnd: ; End of Master messages,
0179 8033     448    SJMP     Dismiss
                                449 ;
                                450 ;
                                451 ;
                                452 ;
                                453 ; Terminate mastership due to an arbitration loss:
                                454
017B         455    Mar1:
                                456
017B 309B02  457    JNB      STR,Mar12 ; If lost arbitration due to other
                                458 ; Master's Start, go be a slave.
017E 014D    459    AJMP     GoSlave
                                460 ;
0180         461    Mar12:
0180 21AE     462    AJMP     Dismiss
                                463 ;
                                464 ;
                                465 ;
                                466 ; Switch from Master to Slave due to arbitration loss after completing

```


Using the 8XC751/752 in multimaster I²C applications AN430

PPCODE1 83C751 Multimaster I2C Routines

4/14/1992 PAGE 10

```

467 ; transmission of a message. The MASTRQ bit was cleared trying to write a
468 ; Stop, and we need to set it again on order to retry transmission when the
469 ; bus gets free again.
470
0182 471 MArlEnd:
0182 D2DE 472 SETB MASTRQ ; Set Master Request – which will get
473 ; into effect when we are done as a
474 ; slave.
0184 217B 475 AJMP MArl
476
477 ; Handling arbitration loss while transmitting an address
478
0186 209BF2 479 AdTxAr1: JB STR,MArl ; Non-synchronous Start or Stop.
0189 209AEF 480 JB STP,MArl
481
482 ; Switch from Master to Slave due to arbitration loss while transmitting
483 ; an address – complete receiving the address transmitted by the new Master.
484
018C B80003 485 CJNE R0,#0,AdTxAr12
486 ; Ar1 on last bit of address
487 ; (R0 is 0 on exit from XmAddr).
018F 14 488 DEC A ; The lsb sent, in which arl occurred
489 ; must have been 1. By decrementing
490 ; A we get the address that won.
0190 8012 491 SJMP AdAr3
492
0192 493 AdTxAr12:
0192 03 494 RR A ; Realign partially Tx'ed ACC
0193 F9 495 MOV R1,A ; and save it in R1
0194 E8 496 MOV A,R0 ; Pointer for lookup table
0195 9001A6 497 MOV DPTR,#MaskTable
0198 93 498 MOVC A,@A+DPTR
0199 59 499 ANL A,R1 ; Set address bits to be received,
500 ; and the bit on which we lost
501 ; arbitration to 0
502 ; Now we are ready to receive the rest
503 ; of the address.
504
019A 759890 506 MOV I2CON,#BCXA+BCARL ; Clear flags and release the clock.
507
019D 5108 508 ACALL RBi3 ; Complete the address using reception
509 ; subroutine.
019F 209D02 510 JB DRDY,AdAr3 ; Around if received address OK
01A2 01B3 511 AJMP SMsgEnd ; Unexpected Start or Stop – end
512 ; as a slave.
01A4 0155 513 AdAr3: AJMP STstRW ; Proceed to check the address
514 ; as a slave.
515
01A6 FF7E3E1E 516 MaskTable: DB Offh,7Eh,3Eh,1Eh,0Eh,06h,02h,00h, ; Offh is dummy
01AA 0E060200
517

```

Using the 8XC751/752 in multimaster I²C applications AN430

PPCODE1 83C751 Multimaster I2C Routines

4/14/1992 PAGE 11

```

518 ; End I2C Interrupt Service Routine:
519
01AE 711E 520 Dismiss: ACALL I2CDONE
521
01B0 7598F4 522 MOV I2CON,#BCARL+BCSTP+BCDR+BCXA+BDLE
01B3 C2DC 523 CLR TIRUN
01B5 D0E0 524 POP ACC
01B7 FA 525 MOV R2,A
01B8 D0E0 526 POP ACC
01BA F9 527 MOV R1,A
01BB D0E0 528 POP ACC
01BD F8 529 MOV R0,A
01BE D0E0 530 POP ACC
01C0 D0D0 531 POP PSW
01C2 D2AC 532 SETB EI2
533
01C4 22 534 RET ; Return from I2C interrupt Service Routine
535
536 ;*****
537 ; Byte Transmit and Receive Subroutines
538 ;*****
539
540
541
542 ; XmAddr: Transmit Address and R/W~
543 ; XmByte: Transmit a byte
544
01C5 F599 545 XmAddr: MOV I2DAT,A ; Send first bit, clears DRDY.
01C7 75981C 546 MOV I2CON,#BCARL+BCSTR+BCSTP
547 ; Clear status, release SCL.
01CA 7808 548 MOV R0,#8 ; Set R0 as bit counter
01CC 8004 549 SJMP XmBit2
01CE 7808 550 XmByte: MOV R0,#8
01D0 F599 551 XmBit: MOV I2DAT,A ; Send the first bit.
01D2 23 552 XmBit2: RL A ; Get next bit.
01D3 309EFD 553 JNB ATN,$ ; Wait for bit sent.
01D6 309D08 554 JNB DRDY,XmBex ; Should be data ready.
01D9 D8F5 555 DJNZ R0,XmBit ; Repeat until all bits sent.
01DB 7598A0 556 MOV I2CON,#BCDR+BCXA ; Switch to receive mode.
01DE 309EFD 557 JNB ATN,$ ; Wait for acknowledge bit.
558 ; flag cleared.
01E1 22 559 XmBex: RET
560
561 ;
562 ; Byte receive routines.
563 ;
564 ; ClsRcv8 clears the status register (from Start condition)
565 ; and then receives a byte.
566 ; AckRcv8 Sends an acknowledge, and then receives a new byte.
567 ; If a Start or Stop is encountered immediately after the
568 ; ack, AckRcv8 returns with 7 in R0.
569 ; ClaRcv8 clears the transmit active state and releases clock

```

Using the 8XC751/752 in multimaster I²C applications AN430

PPCODE1 83C751 Multimaster I2C Routines

4/14/1992 PAGE 12

```

570 ; (from the acknowledge).
571 ;
572 ; A contains the received byte upon return.
573 ; R0 is being used as a bit counter.
574 ;
575
01E2 75989C 576 ClsRcv8: MOV I2CON,#BCARL+BCSTR+BCSTP+BCXA
577 ;Clear status register.
01E5 309EFD 578 JNB ATN,$
01E8 309D22 579 JNB DRDY,RCVex
01EB 800F 580 SJMP Rcv8
581
01ED 759900 582 AckRcv8: MOV I2DAT,#0; Send Ack (low)
01F0 309EFD 583 JNB ATN,$
01F3 309D18 584 JNB DRDY,RCVerr ; Bus exception - exit.
01F6 7598A0 585 ClaRcv8: MOV I2CON,#BCDR+BCXA ; clear status, release clock
586 ;from writing the Ack.
01F9 309EFD 587 JNB ATN,$
588
01FC 7807 589 Rcv8: MOV R0,#7 ; Set bit counter for the first seven
590 ; bits.
01FE E4 591 CLR A ; Init received byte to 0.
01FF 4599 592 RBit: ORL A,I2DAT ; Get bit, clear ATN.
0201 23 593 RBit2: RL A ; Shift data.
0202 309EFD 594 JNB ATN,$ ; Wait for next bit.
0205 309D05 595 JNB DRDY,RCVex ; Exit if not a data bit (could be Start/
596 ; Stop, or bus/protocol error)
0208 D8F5 597 RBit3: DJNZ R0,RBit ; Repeat until 7 bits are in.
020A A29F 598 MOV C,RDAT ; Get last bit, don't clear ATN.
020C 33 599 RLC A ; Form full data byte.
020D 22 600 RCVex: RET
601
020E 7809 602 RCVerr: MOV R0,#9 ; Return non legitimate bit count
0210 22 603 RET
604
605
606 ;*****
607 ; Timer I Interrupt Service Routine
608 ; I2C us Timeout
609 ;*****
610
611 ; In addition to reporting the timeout in MSGSTAT, we update a failure
612 ; counter, TITOCNT. This allows different types of timeout handling by the
613 ; main program.
614
0211 C2DE 615 TIISR: CLR MASTRQ ; "Manual" reset.
0213 759801 616 MOV I2CON,#BXSTP ;
0216 7598BC 617 MOV I2CON,#BCXA+BCDR+BCARL+BCSTR+BCSTP
618
0219 752430 619 TI1: MOV MSGSTAT,#TIMOUT ; Status Flag for Main.
021C 74FF 620 TI2: MOV A,#0FFh ;
021E B52902 621 CJNE A,TITOCNT,TI3 ; Increment TITOCNT, saturating

```

Using the 8XC751/752 in multimaster I²C applications AN430

PPCODE1 83C751 Multimaster I2C Routines

4/14/1992

PAGE 13

```

0221 8002      622      SJMP      TI4      ; at FFh.
0223 0529      623      TI3:      INC      TITOCNT
                                624
0225 5130      625      TI4:      ACALL    RECOVER
                                626
0227 D2DD      627      SETB      CLRTI    ; Clear TI interrupt flag.
0229 114C      628      ACALL      XRETI    ; Clear interrupt pending flag (in
                                629      ; order to re-enable interrupts).
022B 852A81    630      MOV       SP,StackSave ; Realign stack pointer, re-doing
                                631      ; possible stack changes during
                                632      ; the I2C interrupt service routine.
                                633      ; TimerI interrupts in other ISR's
                                634      ; were not allowed !
022E 21AE      635      AJMP      Dismiss    ; Go back to the I2C service routine,
                                636      ; in order to return to the (main)
                                637      ; program interrupted.
                                638
                                639
                                640      ;*****
                                641      ; Bus recovery attempt subroutine
                                642      ;*****
                                643
0230 C2AF      644      RECOVER: CLR      EA
0232 C2DE      645      CLR      MASTRQ ; "Manual" reset.
0234 7598FC    646      MOV      I2CON,#BCXA+BDLE+BCDR+BCARL+BCSTR+BCSTP
0237 C2DF      647      CLR      SLAVEN ; Non I2C TimerI mode
0239 D2DC      648      SETB      TIRUN ; Fire up TimerI. When it overflows, it
                                649      ; will cause I2C interface hardware reset.
023B 79FF      650      MOV      R1,#0ffh
023D 00        651      DLY5:   NOP
023E 00        652      NOP
023F 00        653      NOP
0240 D9FB      654      DJNZ     R1,DLY5
0242 C2DC      655      CLR      TIRUN
0244 D2DD      656      SETB      CLRTI
                                657
0246 D280      658      SETB      SCL      ; Issue clocks to help release other devices.
0248 D281      659      SETB      SDA
024A 7908      660      MOV      R1,#08h
024C C280      661      RC7:     CLR      SCL
024E 00000000  662      DB       0,0,0,0
0252 00
0253 D280      663      SETB      SCL
0255 00000000  664      DB       0,0,0,0
0259 00
025A D9F0      665      DJNZ     R1,RC7
025C C280      666      CLR      SCL
025E 0000      667      DB       0,0
0260 C281      668      CLR      SDA
0262 0000      669      DB       0,0
0264 D280      670      SETB      SCL
0266 00000000  671      DB       0,0,0,0

```

Using the 8XC751/752 in multimaster I²C applications AN430

PPCODE1	83C751 Multimaster I2C Routines	4/14/1992	PAGE 14
026A 00			
026B D281	672 SETB SDA		
026D 00000000	673 DB 0,0,0,0 ; Issue a Stop.		
0271 00			
	674		
0272 7598BC	675 Rex: MOV I2CON,#BCXA+BCDR+BCARL+BCSTR+BCSTP ; clear flags		
0275 D2AF	676 SETB EA		
0277 22	677 RET		
	678		
	679 ;*****		
	680 ;		
	681 ; Main Program		
	682 ;		
	683 ;*****		
	684 ;		
	685 ; Message ping pong game. Each message is transmitted by		
	686 ; a processor that is a master on the I2C bus, and it contains one byte		
	687 ; of data. A processor that receives this data byte as a slave increments		
	688 ; the data by one and transmits it back as a master. The data received is		
	689 ; confirmed to be a one increment of the data formerly sent, unless		
	690 ; it is a "reset" value, chosen to be 00h.		
	691 ; The two participating processors have similar code, where the node		
	692 ; address of the second processor is the destination address of this		
	693 ; one, and vice versa.		
	694 ; The first data byte each processor tries to send is 00h. One of the		
	695 ; processors will acquire the bus first, and the second processor that will		
	696 ; receive this "resetting" 00h will not attempt to confirm it against an		
	697 ; expected value. It will simply increment and transmit it. Subsequent		
	698 ; receptions will be confirmed against the expected value, until 00h data		
	699 ; bytes are sent and the game is effectively reset by the 00h resulting from		
	700 ; the next increment.		
	701 ; A toggling output (TogLED) tells the outer world that the "ping pong"		
	702 ; proceeds well. If something unexpected happens we temporarily activate		
	703 ; another output, ErrLED.		
	704 ; The different tasks of the code are performed in a combination of main-		
	705 ; line program and event routines called from the I2C interrupt service		
	706 ; routine.		
	707		
	708		
	709 ; Initial set-ups:		
	710 ; Load CT1,CT0 bits of I2CFG register, according to the clock		
	711 ; crystal used.		
	712 ; Load RAM location MYADDR with the I2C address of this processor.		
	713 ; We load these values out of ROM table locations (R_CTVAL and R_MYADDR).		
	714 ; One may, instead, load with a MOV <immediate> command.		
	715		
0278 758107	716 Reset: MOV SP,#07h ;Set stack location.		
027B E4	717 CLR A		
027C 90032D	718 MOV DPTR,#R_CTVAL		
027F 93	719 MOVC A,@A+DPTR		
0280 F5D8	720 MOV I2CFG,A ; Load CT1,CT0 (I2C timing, crystal		
	721 ; dependent).		

Using the 8XC751/752 in multimaster I²C applications AN430

PPCODE1 83C751 Multimaster I2C Routines

4/14/1992

PAGE 15

```

0282 E4      722   CLR      A
0283 90032C  723   MOV      DPTR,#R_MYADDR
0286 93      724   MOVC     A,@A+DPTR      ; Get this node's address from ROM table
0287 F525    725   MOV      MYADDR,A      ; into MYADDR RAM location.
              726
0289 C293    727   CLR      OnLED
              728
              729
028B C291    730   Reset2:   CLR      ErrLED ; Flash LED.
028D 51E6    731   ACALL    LDELAY
028F D291    732   SETB     ErrLED
0291 C209    733   CLR      SErrFLAG
0293 C208    734   CLR      TRQFLAG
0295 753750  735   MOV      FAILCNT,#50h
0298 D290    736   SETB     TogLED
029A 753850  737   MOV      TOGCNT,#050h      ; Initialize pin-toggling counter
              738
              739   ; Enable slave operation.
              740   ; The Idle bit is set here for a restart situation - in normal
              741   ; operation this is redundant, as this bit is set upon power_up reset.
029D 759840  742   MOV      I2CON,#IDLE      ; Slave will idle till next Start.
02A0 D2DF    743   SETB     SLAVEN      ; Enable slave operation.
              744
              745   ; Enable interrupts.
              746   ; This is necessary for both Slave and Master operations.
02A2 D2AB    747   SETB     ETI      ; Enable timer I interrupts.
02A4 D2AC    748   SETB     EI2      ; Enable I2C port interrupts.
02A6 D2AF    749   SETB     EA      ; Enable global interrupts.
              750
              751   ; Set up Master operation.
              752
02A8 752000  753   MOV      MASCMD,#0h      ; "Regular" master transmissions.
02AB 90032E  754   MOV      DPTR,#PongADDR
02AE E4      755   CLR      A
02AF 93      756   MOVC     A,@A+DPTR
02B0 F526    757   MOV      DESTADRW,A      ; The partner address. The LSB is
              758   ; low, for a Write transaction.
02B2 752801  759   MOV      MASTCNT,#01h      ; Message length - a single byte.
              760
02B5      761   PPSTART:
02B5 752B00  762   MOV      MasBuf,#00h
              763
              764   ; "Ping" transmission:
              765
02B8      766   PP2:
02B8 D208    767   SETB     TRQFLAG
02BA D2DE    768   SETB     MASTRQ
02BC 79FF    769   MOV      R1,#0ffh
02BE 300809  770   PP22:    JNB      TRQFLAG,PP3      ; Transmitted OK
02C1 D9FB    771   DJNZ     R1,PP22
02C3 D537F2  772   MFAIL1:  DJNZ     FAILCNT,PP2
02C6 5130    773   ACALL    RECOVER

```

PPCODE1 83C751 Multimaster I2C Routines

4/14/1992

PAGE 16

```

02C8 80C1      774      SJMP      Reset2
                775
                776      ; "Pong" reception:
                777
02CA 78FF      778      PP3:      MOV      R0,#0fh ; Software timeout loop count.
02CC 79FF      779      PP31:     MOV      R1,#0fh
02CE 2008E7    780      PP32:     JB       TRQFLAG,PP2 ; Rcvd ok as slave, go transmit.
02D1 200908    781      JB       SErrFLAG,PP5
02D4 D9F8      782      DJNZ     R1,PP32
02D6 D8F4      783      DJNZ     R0,PP31
02D8 5130      784      PPTO:     ACALL    RECOVER ; Software timeout.
02DA 418B      785      AJMP     Reset2
                786
02DC C291      787      PP5:      CLR      ErrLED ; Receive error.
02DE 51E6      788      ACALL    LDELAY
02E0 D291      789      SETB     ErrLED
02E2 C209      790      CLR      SErrFLAG
02E4 41B5      791      AJMP     PPSTART
                792
02E6 7A30      793      LDELAY:     MOV      R2,#030h
02E8 79FF      794      LDELAY1:    MOV      R1,#0fh
02EA D9FE      795      DJNZ     R1,$
02EC DAFA      796      DJNZ     R2,LDELAY1
02EE 22        797      RET
                798
                799      ;*****
                800      ; Slave and Master Event Routines.
                801      ;*****
                802
                803      ;
                804      ;Invoked upon completion of a message transaction.
                805      ;This is the part of the application program actually dealing
                806      ;with the data communicated on the I2C bus, by responding to
                807      ;new data received and/or preparing the next transaction.
                808
                809
                810      ; Slave Event Routines
                811      ;
                812      ; These routines are invoked by the I2C interrupt service routine when a
                813      ; message transaction as a slave has been completed. Our "application":
                814      ; reacts to a message received as a slave with the routine SRCvdR.
                815      ; The calls that indicate erroneous reception are treated the same way as
                816      ; erroneous data reception in the "ping pong" game.
                817
                818      ;SRCvdR
                819      ;Invoked when a new message has been received as a Slave.
                820
02EF 00      821      SRCvdR:     NOP
02F0 E52F      822      MOV      A,SRcvBuf
02F2 7005      823      JNZ      SR2
02F4 752B01    824      MOV      MasBuf,#01h ; It was ping-pong reset value
02F7 800F      825      SJMP     SR3

```

Using the 8XC751/752 in multimaster I²C applications

PPCODE1 83C751 Multimaster I2C Routines

4/14/1992 PAGE 17

```

826
02F9 052B      827      SR2:      INC      MasBuf ; The expected data.
02FB B52B0F    828      CJNE     A,MasBuf,ErrSR
02FE 052B      829      INC      MasBuf ; Data for next transmission – the data
830              ; received incremented by 1.
831
832              ;A successful two way data exchange. Let the outside world know by
833              ;toggling an output pin driving a LED. We actually toggle only
834              ;when a number of such exchanges is completed, in order to
835              ;slow down the changes for a good visual indication.
836
0300 D53805    837      DJNZ     TOGCNT,SR3
0303 B290      838      CPL      TogLED ; Toggle output
0305 753850    839      MOV     TOGCNT,#050h ;
840
0308 C209      841      SR3:      CLR      SErrFLAG
030A D208      842      SETB    TRQFLAG ; Request main to transmit
030C 22        843      RET
844
030D D209      845      ErrSR:    SETB    SErrFLAG
030F 22        846      RET
847
848
849      ;SLnRcvdR
850      ;Invoked when a message received as a Slave is too long
851      ;for the receive buffer.
852
853      ;STXedR
854      ;Invoked when a Slave completed transmission of its buffer.
855      ;We do not expect to get here, since we do not plan to have
856      ;in our system a master that will request data from this node.
857      ;
858      ;
859      ;SRErrR
860      ;Slave error event subroutine.
861      ;In most applications it will not be used.
862      ;
863
0310          864      SLnRcvdR:
0310          865      STXedR:
0310 80FB      866      SRErrR:    JMP      ErrSR
867
868
869      ;
870      ;MastNext – Master Event Routine.
871      ;
872      ;Invoked when a Master transaction is completed, or terminated
873      ;"willingly" due to lack of acknowledge by a slave.
874      ;
875
0312          876      MastNext:
0312 E524      877      MOV     A,MSGSTAT

```

Using the 8XC751/752 in multimaster I²C applications AN430

PPCODE1 83C751 Multimaster I2C Routines

4/14/1992 PAGE 18

```

0314 B42206 878 CJNE      A,#MTXED,MN1
0317 753750 879 MOV       FAILCNT,#50h
031A C208 880 CLR       TRQFLAG
031C 22 881 RET
031D 882 MN1:
031D 22 883 RET
884
885 ;I2CDONE
886 ;Called upon completion of the I2C interrupt service routine.
887 ;In this example it monitors exceptions, and invokes the bus
888 ;recovery routine when too many occurred.
889
031E 890 I2CDONE:
031E E524 891 MOV       A,MSGSTAT
0320 B43208 892 CJNE      A,#NOTSTR,I2CD1
0323 D53705 893 DJNZ      FAILCNT,I2CD1
0326 753701 894 MOV       FAILCNT,#01h ; Too many "illegal" i2c interrupts
0329 C2AC 895 CLR       EI2 ; - shut off.
032B 22 896 I2CD1:    RET
897
898
899 ;*****
900 ; I2C Communications Table:
901 ;*****
902
903
904
905 ; We used table driven values for clarity, one may use immediate to load
906 ; these values and save several lines of code.
907
908 ; Contents is used in the beginning of the main program to load
909 ; RAM location MYADDR and the I2CFG register.
910 ; The node address, in R_MYADDR, is application specific, and unique for
911 ; each device in the I2C network.
912 ; R_CTVAL depends on the crystal clock frequency.
913
032C 4E 914 R_MYADDR: DB 4Eh ; This node's address
915
032D 02 916 R_CTVAL: DB 02h ; CT1, CT0 bit values
917
918 ;*****
919 ; Application Code Definitions
920 ;*****
921
032E 4A 922 PongADDR: DB 4Ah ; The address of the "partner" in
923 ; the ping-pong game.
924
925
926
927
928 END
929

```

VERSION 1.2h ASSEMBLY COMPLETE, 0 ERRORS FOUND

Using the 8XC751/752 in multimaster I²C applications AN430

PPCODE1 83C751 Multimaster I2C Routines

4/14/1992 PAGE 19

ACC	D ADDR	00E0H	PREDEFINED
ACKRCV8	C ADDR	01EDH	
ADAR3	C ADDR	01A4H	
ADDRRCV	C ADDR	0050H	NOT USED
ADTXARL	C ADDR	0186H	
ADTXARL2	C ADDR	0192H	
APPFLAGS	D ADDR	0021H	
ARL	B ADDR	009CH	PREDEFINED
ATN	B ADDR	009EH	PREDEFINED
BCARL	NUMB	0010H	
BCDR	NUMB	0020H	
BCSTP	NUMB	0004H	
BCSTR	NUMB	0008H	
BCXA	NUMB	0080H	
BIDLE	NUMB	0040H	
BMRQ	NUMB	0040H	NOT USED
BTIR	NUMB	0010H	NOT USED
BXSTP	NUMB	0001H	
BXSTR	NUMB	0002H	
CLARCV8	C ADDR	01F6H	
CLRTI	B ADDR	00DDH	PREDEFINED
CLSRCV8	C ADDR	01E2H	
DESSUBAD	D ADDR	0027H	
DESTADRW	D ADDR	0026H	
DISMISS	C ADDR	01AEH	
DLY5	C ADDR	023DH	
DRDY	B ADDR	009DH	PREDEFINED
EA	B ADDR	00AFH	PREDEFINED
EI2	B ADDR	00ACH	PREDEFINED
ERRLED	B ADDR	0091H	
ERRSR	C ADDR	030DH	
ETI	B ADDR	00ABH	PREDEFINED
FAILCNT	D ADDR	0037H	
GM2	C ADDR	00CEH	
GM3	C ADDR	00D0H	
GM4	C ADDR	00E3H	
GM5	C ADDR	00E5H	
GM6	C ADDR	010BH	
GM7	C ADDR	0113H	
GM8	C ADDR	0115H	
GOIDLE	C ADDR	00BBH	

PPCODE1 83C751 Multimaster I2C Routines

GOMAS2	C ADDR	00D8H	PREDEFINED
GOMASTER	C ADDR	00BDH	
GOSLAVE	C ADDR	004DH	
I2CD1	C ADDR	032BH	NOT USED
I2CDONE	C ADDR	031EH	
I2CFG	D ADDR	00D8H	PREDEFINED
I2CISR	C ADDR	0023H	NOT USED
I2CON	D ADDR	0098H	PREDEFINED
I2DAT	D ADDR	0099H	PREDEFINED
LDELAY	C ADDR	02E6H	
LDELAY1	C ADDR	02E8H	
MARL	C ADDR	017BH	
MARL2	C ADDR	0180H	
MARLEND	C ADDR	0182H	
MASBUF	D ADDR	002BH	
MASCMD	D ADDR	0020H	NOT USED
MASKTABLE	C ADDR	01A6H	NOT USED
MASTCNT	D ADDR	0028H	
MASTER	B ADDR	0099H	PREDEFINED
MASTNEXT	C ADDR	0312H	
MASTRQ	B ADDR	00DEH	PREDEFINED
MFALL1	C ADDR	02C3H	NOT USED
MGO	NUMB	0020H	
MMSGEND	C ADDR	0179H	
MN1	C ADDR	031DH	
MRCV	C ADDR	0139H	
MRCV2	C ADDR	013FH	PREDEFINED
MRCVED	NUMB	0021H	PREDEFINED
MRCVLOOP	C ADDR	013DH	PREDEFINED
MSGSTAT	D ADDR	0024H	
MTX	C ADDR	011AH	
MTXED	NUMB	0022H	PREDEFINED
MTXLOOP	C ADDR	011BH	
MTXNAK	NUMB	0023H	
MTXNOSLV	NUMB	0024H	
MTXSTOP	C ADDR	0154H	
MTXSTOP2	C ADDR	015CH	
MTXSTOP3	C ADDR	0163H	
MYADDR	D ADDR	0025H	
NOACK	C ADDR	0134H	
NOGO	C ADDR	0047H	

PPCODE1	83C751	Multimaster	I2C	Routines	4/14/1992	PAGE 20
ACC	D ADDR	00E0H				
ACKRCV	C ADDR	01EDH				
ADAR3	C ADDR	01A4H				
ADDRCV	C ADDR	0030H				
ABTXARL	C ADDR	0184H				
ABTXARL2	C ADDR	0183H				
APPLAGS	D ADDR	0031H				
ARL	B ADDR	0098H				
ATN	B ADDR	0099H				
BCARL	NUMB	0010H				
BODR	NUMB	0020H				
BCSTP	NUMB	0004H				
BCSTR	NUMB	0008H				
BCKA	NUMB	0008H				
BIDLE	NUMB	0040H				
BMRO	NUMB	0040H				
BTR	NUMB	0010H				
BRSTP	NUMB	0001H				
BRSTR	NUMB	0003H				
CLARCV	C ADDR	01F4H				
CLRT	B ADDR	0000H				
CLSCV	C ADDR	01E3H				
DESSUBAD	D ADDR	0037H				
DESTADRW	D ADDR	0038H				
DEMISS	C ADDR	01AEH				
DRDY	C ADDR	033DH				
EA	B ADDR	00AFH				
EL	B ADDR	00ACH				
ERRLED	B ADDR	0081H				
ERRR	C ADDR	0080H				
ETI	B ADDR	008BH				
FAILCNT	D ADDR	0037H				
GMS	C ADDR	00CEH				
GMS	C ADDR	00D0H				
GMS	C ADDR	00E3H				
GMS	C ADDR	00E2H				
GMS	C ADDR	0108H				
GMS	C ADDR	0113H				
GMS	C ADDR	0115H				
GMS	C ADDR	008BH				

AN430

4/14/1992 PAGE 21

Using the 8XC751/752 in multimaster I²C applications AN430

PPCODE1 83C751 Multimaster I2C Routines

SR2.	C ADDR	02F9H	
SR3.	C ADDR	0308H	
SRCV2	C ADDR	005DH	NOT USED
SRCV3	C ADDR	0068H	PREDEFINED
SRCVBUF.	D ADDR	002FH	
SRCVD	NUMB	0011H	
SRCVDR	C ADDR	02EFH	
SRCVEND.	C ADDR	0076H	
SRCVERR.	C ADDR	00A7H	
SRCVSTO.	C ADDR	0066H	
SRERR	NUMB	0014H	
SRERRR	C ADDR	0310H	
SRLNG	NUMB	0012H	NOT USED
STACKSAVE	D ADDR	002AH	PREDEFINED
STP.	B ADDR	009AH	PREDEFINED
STR.	B ADDR	009BH	PREDEFINED
STSTRW	C ADDR	0055H	
STX2	C ADDR	0085H	NOT USED
STXBUF.	D ADDR	0033H	
STXED	NUMB	0013H	
STXEDR	C ADDR	0310H	
STXERR	C ADDR	00AEH	NOT USED
STXLP	C ADDR	0093H	PREDEFINED
SUBADD.	B ADDR	0000H	
TI1.	C ADDR	0219H	NOT USED
TI2.	C ADDR	021CH	NOT USED
TI3.	C ADDR	0223H	NOT USED
TI4.	C ADDR	0225H	
TIISR	C ADDR	0211H	
TIMERI	C ADDR	001BH	NOT USED
TIRUN	B ADDR	00DCH	PREDEFINED
TITOCNT.	D ADDR	0029H	PREDEFINED
TOGCNT.	D ADDR	0038H	
TOGLED	B ADDR	0090H	
TRQFLAG.	B ADDR	0008H	
XMADDR.	C ADDR	01C5H	PREDEFINED
XMBEX	C ADDR	01E1H	
XMBIT	C ADDR	01D0H	
XMBIT2	C ADDR	01D2H	
XMBYTE	C ADDR	01CEH	
XRETI	C ADDR	004CH	PREDEFINED

4/14/1992 PAGE 22

PPCODE1 83C751 Multimaster I2C Routines	
SR2.	C ADDR 02F9H
SR3.	C ADDR 0308H
SRCV2	C ADDR 005DH
SRCV3	C ADDR 0068H
SRCVBUF.	D ADDR 002FH
SRCVD	NUMB 0011H
SRCVDR	C ADDR 02EFH
SRCVEND.	C ADDR 0076H
SRCVERR.	C ADDR 00A7H
SRCVSTO.	C ADDR 0066H
SRERR	NUMB 0014H
SRERRR	C ADDR 0310H
SRLNG	NUMB 0012H
STACKSAVE	D ADDR 002AH
STP.	B ADDR 009AH
STR.	B ADDR 009BH
STSTRW	C ADDR 0055H
STX2	C ADDR 0085H
STXBUF.	D ADDR 0033H
STXED	NUMB 0013H
STXEDR	C ADDR 0310H
STXERR	C ADDR 00AEH
STXLP	C ADDR 0093H
SUBADD.	B ADDR 0000H
TI1.	C ADDR 0219H
TI2.	C ADDR 021CH
TI3.	C ADDR 0223H
TI4.	C ADDR 0225H
TIISR	C ADDR 0211H
TIMERI	C ADDR 001BH
TIRUN	B ADDR 00DCH
TITOCNT.	D ADDR 0029H
TOGCNT.	D ADDR 0038H
TOGLED	B ADDR 0090H
TRQFLAG.	B ADDR 0008H
XMADDR.	C ADDR 01C5H
XMBEX	C ADDR 01E1H
XMBIT	C ADDR 01D0H
XMBIT2	C ADDR 01D2H
XMBYTE	C ADDR 01CEH
XRETI	C ADDR 004CH

I²C slave routines for the 83C751

AN433

Author: Greg Goodhue

The S83C751/S87C751 Microcontroller combines in a small package the benefits of a high-performance microcontroller with on-board hardware supporting the Inter-Integrated Circuit (I²C) bus interface.

The 8XC751 can be programmed both as an I²C bus master, a slave, or both. An overview of the I²C bus and description of the bus support hardware in the 8XC751 microcontrollers appears in application note AN422, "Using the 8XC751 Microcontroller as an I²C Bus Master." That application note includes a programming example, demonstrating a bus-master code. Here we show an example of programming the microcontroller as an I²C slave.

The code listing demonstrates communications routines for the 8XC751 as a slave on the I²C bus. It compliments the program in AN422 which demonstrates the 8XC752 as an I²C bus master. One may demonstrate two 8XC751 devices communicating with each other on the I²C bus, using the AN422 code in one, and the program presented here in the other. The examples presented here and in AN422 allow the 751 to be either a master or a slave, but not both. Switching between master and slave roles in a multimaster environment is described in application note AN435.

The software for a slave on the bus is relatively simple, as the processor plays a relatively passive role. It does not initiate bus transfers on its own, but responds to a master initiating the communications. This is true whether the slave receives or transmits data—transmission takes place only as a response to a bus master's request. The slave does not have to worry about arbitration or about devices which do not acknowledge their address. As the slave is not supposed to take control of the bus, we do not demand it to resolve bus exceptions or "hangups". If the bus becomes inactive the processor simply withdraws, not interfering with the master (or masters) on the bus which should (hopefully) try to resolve the situation.

The 8XC751 has a single bit I²C hardware interface where the registers may directly affect the levels on the bus, and the software interacting with the hardware registers takes part in the protocol implementation. The hardware and the low level routines dealing with the registers are tightly coupled. We repeat here the warning from the 751 bus-master application note: one should take extra care if trying to modify these lower level routines.

The service routine for the I²C slave is interrupt driven per message. This allows for master communication requests which are

not synchronized with the application program running on the slave. It is possible to write simple slave application programs which will not be interrupt driven, taking care not to lose master transmissions while doing something else, but the user should be discouraged from doing so. As the slave should respond to asynchronous requests of masters on the bus, an interrupt driven service routine makes sense—and, as the code demonstrates, is simple to implement.

DEMONSTRATION CODE

The main program operation, intended for demonstration only, is simple. There are two data buffers, one for data reception and one for data transmission. When new data has been received from the I²C bus into the receive buffer, the program writes it into the transmit buffer. The first and second bytes of received data are also copied to Port 1 and Port 3, respectively. When a bus master requests to read data, Port 1 and Port 3 will be returned for the first two bytes of requested data, while the remaining bytes will come from the transmit buffer. This allows for simple testing of a master and slave system by having the master compare data received to data sent. This scheme also allows the 8XC751 to be used as a two-byte I²C I/O port.

The program begins at address 0, where the microprocessor begins execution after a hardware reset. This location contains a jump instruction to the main program, which starts at the label Reset (towards the end of the listing). Upon reset, the program initializes the stack pointer, the I²C address of the slave processor (MyAddr) and clears the data buffers and software flags. In this program the receive and transmit buffers are each eight bytes long—the maximum number of bytes is defined by the label MaxBytes. One may easily change the program to handle longer messages by changing the value of MaxBytes and allocating more data memory to the buffers.

The I²C interface is configured to operate as a slave by setting the msb of register I²CFG. This is done simultaneously with loading the appropriate value of CTVAL—bits CT0 and CT1, which are determined by the frequency of the microprocessor's crystal. The interface hardware is explicitly instructed to get into the slave idle mode by setting the appropriate bit in the I2CON register. Timer 1, which operates as a "watchdog" timer detecting bus hangups, is activated and its interrupts are enabled.

After the initialization, the program gets to the label MainLoop. Most of the time the program will "hang" in a wait loop at this label, simply

waiting for an I²C interrupt to occur. When there is an I²C bus request there will be an interrupt, the service routine will be executed and we shall return to the MainLoop label. If the service routine receives new data, it sets a flag, DatFlag, signalling that data has been updated. This flag will allow us to leave the MainLoop label, and execute a short routine copying the updated input buffer to the output (transmit) buffer.

If a new bus interrupt comes before overwriting of the old read buffer data is completed, and an undesirable "mix" of old and new data might occur. This type of situation is avoided by disabling the I²C interrupts (clearing the IE2 bit in the Interrupt Enable Register) just before copying the data to the transmit buffer, and re-enabling the interrupts when the copy operation is completed.

When the copy routine is completed the DatFlag is cleared and we jump back to MainLoop, waiting for the next interrupt to occur. If the interrupt is for data transmission the service routine will not set DatFlag, and upon return we shall remain at the MainLoop label.

THE INTERRUPT SERVICE ROUTINE

The service routine is interrupt driven with respect to the start of each I²C frame, but within each frame the interaction with the hardware is based on polling. An occurrence of a Start on the bus will cause an interrupt that will initiate the service routine which starts at address 23H. After saving registers, all interrupts except the I²C interrupt itself are enabled, as we want to allow response to other interrupts during the routine. The philosophy behind this is that the I²C may be a lower priority than some other operations in the system. Since the I²C hardware will stretch the clock until the program responds, an interrupt of reasonable duration will not have a harmful effect on the data transfer.

Since we intend to react to the I²C hardware by polling the ATN flag in wait loops, we do not want the expected changes on the bus to take us again to the beginning of the routine. Therefore, the EI2 flag is cleared, masking further I²C interrupts even when interrupts are re-enabled (by the ACALL to a RETI instruction).

At the label Slave, the routine starts receiving the address on the bus. Each new address bit is read after a software wait loop detects that the ATN flag is set by the hardware. Note that with the single bit implementation of the I²C port on the 8XC751 the software must

I²C slave routines for the 83C751

AN433

closely support the hardware: for example, we need to explicitly clear the Start status before we enter a wait loop for the next bit. If the software does not clear the Start flag, the hardware will stretch the low period of the clock (SCL line) on the bus—and the first address bit will simply not occur. (Such a state will not go on forever—eventually the processor will release the bus as a result of a Timer 1 timeout.)

Reception of the eight bits of Address + R/W is completed using part of the receive byte subroutines. The address received is compared to MyAddr, the address of this specific slave. If the address is different the processor goes idle and leaves the service routine. If the message is intended for this processor (received address matches MyAddr) the Read/Write bit is tested, and the program jumps to the appropriate labels. When the R/W bit is low the master requests a Write—and this slave should receive the data written into it. When the R/W bit is high the master is requesting a Read and this slave should transmit the data (at code label Read).

For "Master Write" we send an acknowledge for the address byte and proceed with receiving the data bytes, responding with an acknowledge for each and transferring them into the receive buffer. For long messages, when the buffer is full (we have received MaxByte bytes) we read from the bus one additional byte and then send a negative acknowledge, letting the master know it

should stop sending us data. Then we set DatFlag to signal the mainline program that new data has been received, and jump to MsgEnd. At the MsgEnd label we wait for the next Stop or Repeated Start. On a Stop we resume the idle mode (Goldle) and return from the service routine. On a Restart the slave process starts again with reception of the new address at the label Slave.

If the message is short enough so that the receive buffer is not filled up, the RcvByte subroutine (called after WrtLoop) will return due to the Stop condition, DRDY will not be set, and we shall exit the loop via label WLEX—setting the DatFlag and proceeding to MsgEnd.

For "Master Read" the transmit buffer is sent on the bus byte by byte in the RdLoop, using the XmitByte subroutine. We exit the loop when all the buffer is transmitted, or the Master does not respond with an acknowledge. Note that lack of acknowledgement for slave transmission does not necessarily indicate a problem or that the receiving master is busy. This could very well be a normal operation of the protocol, which defines that a receiving master signals the transmitting slave to end its message by explicitly transmitting a negative acknowledge as a response to the last byte the master is interested in. The protocol does not include inherent means for specifying in advance the length of a requested message.

SUBROUTINES

The lower level subroutines closely interact with the hardware and the activity on the bus. The XmitByte subroutine transmits one byte and receives the acknowledge bit that comes in response. The byte receive routine, which one may use from different entry points, receives a data or an address byte, and takes care of acknowledgements. When a Start or Stop is detected the subroutine returns immediately—the calling routine is expected to check the flags to determine whether a whole byte has been received (DRDY will be set), or a Start or a Stop condition has occurred.

Close inspection of RcvByte code shows that a total of nine bits are being read off the bus. The first bit does not belong to the received byte, but is the acknowledge this processor sent in response of the former byte or address. Reading the Ack bit from the I2DAT register clears the Transmit Active state and DRDY, thus releasing SCL and allowing the bus activity to proceed to the next data bit. Upon return the Ack bit is left in the Carry flag, and the actual data byte received is returned in the Acc register.

Upon Timer 1 interrupt code execution commences at address 1BH, where there is a jump to the service routine Timer1. This interrupt is caused by the watchdog timer, as a result of an I²C bus that is "hanging" without activity in the middle of a transmission for too long a period of time. The slave simply clears the bus interface, and starts all over again at the label Reset.

I²C slave routines for the 83C751

AN433

```

ByteCnt    DATA    22h    ; Send/receive byte counter.
TDAT       DATA    23h    ; Temporary holding register.
MyAddr     DATA    24h    ; Holds address of THIS slave.

AdrRcvd    DATA    25h    ; Holds received slave address + R/W.
RWFlag     BIT       AdrRcvd.0 ; Slave read/write flag.

;*****
;
; Begin Code
;*****

; Reset and interrupt vectors.
                AJMP     Reset    ; Reset vector at address 0.

; A timer I timeout usually indicates a 'hung' bus.

                ORG      1Bh      ; Timer I (I2C timeout) interrupt.
                AJMP     TimerI

; I2C interrupt is used to detect a start while the slave is idle.

                ORG      23h      ; I2C interrupt.
                PUSH     PSW      ; Save status.
                PUSH     ACC      ; Save accumulator.
                CLR      ES      ; Disable I2C interrupt.
                ACALL    ClrInt   ; Re-enable interrupts.

;*****
;
; Main Transmit and Receive Routines
;*****

Slave:         MOV      I2CON,#BCARL+BCSTP+BCSTR+BCXA ; Clear start status.
                JNB      ATN,$    ; Wait for next data bit.
                MOV      BitCnt,#7 ; Set bit count.

                ACALL    RcvB2     ; Get remainder of slave address.
                MOV      AdrRcvd,A ; Save received address + R/W bit.
                CLR      ACC.0
                CJNE     A,MyAddr,GoIdle ; Enter idle mode if not our address.

                JB       RWFlag,Read ; Read or Write?
                MOV      R0,#RcvDat ; Set up receive buffer pointer.
                MOV      ByteCnt,#MaxBytes ; Max 4 bytes can be received.

WrtLoop:       ACALL    SendAck    ; Send acknowledge.
                ACALL    RcvByte   ; Get data byte from master.
                JNB      DRDY,WLex  ; Must be end of frame?
                MOV      @R0,A     ; Save data.
                INC      R0        ; Advance buffer pointer.
                DJNZ     ByteCnt,WrtLoop ; Back to receive if buffer not full.
                ACALL    SendAck    ; Send acknowledge.
                ACALL    RcvByte   ; Get, but do not store add'l data.
                MOV      I2DAT,#80h ; Send negative acknowledge.
                JNB      ATN,$      ; Wait for acknowledge sent.
WLex:          SETB     DatFlag    ; Flag main that data has been received.
                SJMP     MsgEnd    ; Buffer full, enter idle mode.

Read:         MOV      R0,#XmtDat ; Set up transmit buffer pointer.
                MOV      ByteCnt,#MaxBytes ; Max bytes to be sent.

```

I²C slave routines for the 83C751I²C slave routines for the 83C751 AN433

```

ACALL SendAck          ; Send address acknowledge.

RdLoop: MOV     A,@R0          ; Get data byte from buffer.
        CJNE    R0,#XmtDat,RdL1 ; Return port 1 value instead of buffer.
        MOV     A,P1          ; data if this is buffer address 0.
RdL1:    CJNE    R0,#XmtDat+1,RdL2 ; Return port 3 value instead of buffer.
        MOV     A,P3          ; data if this is buffer address 1.

RdL2:    INC     R0            ; Advance buffer pointer.
        ACALL   XmitByte      ; Send data byte.
        JB      NoAck,RLEx    ; Exit if NAK.
        DJNZ    ByteCnt,RdLoop ; Back if more data requested & avail.
RLEx:    SJMP    MsgEnd       ; Done, enter idle mode.

MsgEnd:   JNB     ATN,$         ; Wait for stop or repeated start.
        JB      STR,Slave      ; If repeated start, go to slave mode,
                                ; else enter idle mode.

GoIdle:   MOV     I2CON,#BCSTP+BCXA+BCDR+BCARL+BIDLE ; Enter slave idle mode.
        POP     ACC            ; Restore accumulator.
        POP     PSW            ; Restore status.
        SETB    ES            ; Re-enable I2C interrupts.
        RET

;*****
;                               Subroutines
;*****
; Byte transmit routine.
; Enter with data in ACC.

XmitByte: MOV     BitCnt,#8      ; Set 8 bits of data count.
XmBit:    MOV     I2DAT,A         ; Send this bit.
        RL       A              ; Get next bit.
        JNB     ATN,$           ; Wait for bit sent.
        DJNZ    BitCnt,XmBit    ; Repeat until all bits sent.
        MOV     I2CON,#BCDR+BCXA ; Switch to receive mode.
        JNB     ATN,$           ; Wait for acknowledge bit.
        MOV     Flags,I2DAT      ; Save acknowledge bit.
        RET

; Byte receive routines.
; SendAck : sends an I2C acknowledge.
; RcvByte : receives a byte of data.
; RcvB2 : receives a partial byte of I2C data, used to allow reception of
;         7 bits of slave address information.
; Data is returned in the ACC.

SendAck:  MOV     I2DAT,#0        ; Send receive acknowledge.
        JNB     ATN,$           ; Wait for acknowledge sent.
        RET

RcvByte:  MOV     BitCnt,#8      ; Set bit count.
RcvB2:    CLR     A              ; Init received byte to 0.
RBit:     ORL     A,I2DAT        ; Get bit, clear ATN.
        RL       A              ; Shift data.
        JNB     ATN,$           ; Wait for next bit.
        JNB     DRDY,RBEx       ; Exit if not a data bit.
        DJNZ    BitCnt,RBit     ; Repeat until 7 bits are in.
        MOV     C,RDAT          ; Get last bit, don't clear ATN.
        RLC     A              ; Form full data byte.
RBEx:     RET

```

```

; Timer I timeout interrupt service routine.
TimerI:  SETB    CLR TI      ; Clear timer I interrupt.
         MOV     I2CFG,#0    ; Turn off I2C.
         MOV     I2CON,#BCXA+BCDR+BCARL+BCSTR+BCSTP ; Reset I2C flags.
         ACALL   CtrInt      ; Clear interrupt pending flag.
         AJMP    Reset       ; Return to mainline.

CtrInt:  RETI

; *****
; ***** Main Program *****
; *****

Reset:   MOV     SP,#2Fh      ; Set stack location.
         MOV     IE,#90h     ; Enable I2C interrupt.

         MOV     R0,#RcvDat   ; Set up pointer to data area.
         MOV     R1,#2*MaxBytes ; Set up buffer length counter.
RLoop:   MOV     @R0,#0        ; Clear buffer memory.
         INC     R0           ; Advance to next buffer position.
         DJNZ    R1,RLoop     ; Repeat until done.

         MOV     MyAddr,#40h   ; Set slave address.
         MOV     Flags,#0      ; Clear system flags.
         MOV     I2CFG,#80h+CTVAL ; Enable slave functions.
         MOV     I2CON,#BIDLE  ; Put slave into idle mode.
         SETB    ETI          ; Enable timer I interrupts.
         SETB    TIRUN        ; Turn on timer I.

; This sample mainline program copies the first two received bytes to Port 1
; and Port 3 whenever there is an I2C write operation. It also copies the
; rest of the input buffer to the output buffer at the same time.

MainLoop: JNB     DatFlag,$    ; Wait for data sent from I2C.
         CLR     EA           ; Turn off interrupts during data move.

         MOV     P1,RcvDat+1   ; First buffer location goes to port 1.
         MOV     P3,RcvDat+1   ; Second buffer location goes to port 3.

         MOV     R0,#RcvDat     ; Set input buffer start pointer.
         MOV     R1,#XmtDat     ; Set output buffer start pointer.
         MOV     R2,#MaxBytes   ; Set buffer length counter.
ML2:     MOV     A,@R0          ; Get data from input buffer.
         MOV     @R1,A          ; Store data in output buffer.
         INC     R1             ; Increment input buffer pointer.
         INC     R0             ; Increment output buffer pointer.
         DJNZ    R2,ML2         ; Repeat until entire buffer is updated.
         CLR     DatFlag       ; Clear I2C transmission flag.

         SETB    EA           ; Data move done, re-enable interrupts.
         SJMP    MainLoop      ; Wait for next I2C transmission.

         END

```

Connecting a PC keyboard to the I²C-bus AN434

CONNECTING A PC KEYBOARD TO THE I²C BUS

This application note illustrates the use of a low-cost 8-bit microcontroller—the 8XC751—to interface a standard PC/AT keyboard to the I²C bus. The 8XC751 (83C751 = ROM-version, 87C571 = EPROM-version) is ideally suited for the task thanks to its built-in I²C interface, small form-factor (24-pin DIP or 28-pin PLCC) and low power consumption (11mA typical @ 12 MHz; see Figure 1). The application software easily fits within the 2K bytes code and 64 bytes data memory provided on the 8XC751.

The PC/AT Keyboard

The PC/AT keyboard transmits data in a clocked serial format consisting of a start bit, 8 data bits (LSB first), an odd parity bit and a stop bit as shown in Figure 2. Besides clock and data, the 5-pin connector (Figure 3) also includes power, ground and a no connect. Note that the PS/2 keyboard interface is logically equivalent, though it uses a different connector. (A sixth pin provides an additional no connect).

When a key is pressed, the PC/AT keyboard transmits a 'make' code and, when the key is released, a 'break' code. The make code consists of an 8-bit 'scan' code denoting the key pressed. The 'break' code (key released) consists of the same 8-bit scan code preceded by a special code—0F0H.

A notable difference from a regular ASCII keyboard is the way SHIFT, CTRL, ALT, etc. control keys work. For an ASCII keyboard, the control keys directly modify the code output. For example, a 61H (ASCII code for 'a') is output if the 'A' key is pressed by itself, while a 41H (ASCII code for 'A') is output if the SHIFT and 'A' keys are pressed simultaneously.

The PC/AT keyboard handles such a key combination as two separate key presses, i.e., SHIFT-MAKE, 'A'-MAKE, SHIFT-BREAK, 'A'-BREAK. The 'A' scan code (1CH) is the same for both the shifted and unshifted state. To determine whether the 'A' scan code is interpreted as 'A' or 'a' the PC must keep track of the presence or absence of a prior SHIFT-MAKE.

Keyboard-to-I²C Hardware (Figure 4)

The 8XC751 on-chip I²C interface allows direct connection of the SDA (Serial Data) and SCL (Serial Clock) pins to the corresponding I²C bus lines. Since the I²C bus is open collector (allowing multimasters), 10K resistors are used to pull the lines to the idle state between keypresses.

The PC/AT keyboard interface is equally simple. The CLK output from the keyboard is used to generate an interrupt (INT0). In response, the 8XC751 interrupt service routine samples the keyboard serial DATA connected to port 0 bit 2 (P0.2).

When used with a PC, the keyboard implements a bidirectional communication protocol by exploiting the fact that both the keyboard and PC can drive the open collector CLK and DATA lines. However, bidirectional communication is not required for basic keyboard operation and in this application, the keyboard is treated as an 'input-only' device.

Keyboard-to-I²C Software

The keyboard-to-I²C software performs three major functions:

- Capture the clocked serial data from the keyboard
- Translate the keyboard data to the corresponding ASCII code
- Send the ASCII code as an I²C message.

When a key is pressed, the CLK output from the keyboard generates an interrupt via INT0. The 8XC751 shifts in the DATA from the keyboard on P0.2 (port 0, bit 2) and extracts the 8-bit scan code from the 11-bit packet.

Next, the scan code is interpreted and converted to the corresponding ASCII code using a look-up table. Keyboard multi-code outputs are converted to single ASCII codes by tracking the state (i.e. shifted vs. unshifted) of the keyboard and using separate look-up tables for each. For

example, a keyboard SHIFT-MAKE, 'A'-MAKE, SHIFT-BREAK, 'A'-BREAK sequence is converted to the ASCII code for uppercase 'A' (41H). The flowchart in Figure 5 depicts the keyboard data capture and code conversion process.

The 8XC751 operates as an I²C slave. When the master issues a read command, the 8XC751 returns the converted ASCII character. The seven least significant bits are used for the ASCII code, while the most significant bit is used as a NEW flag (0 = new, 1 = old). The key code remains marked as new until the master issues a write to the 8XC751 at which point it is marked as old and will be overwritten by the next key processed.

The keyboard-to-I²C software is shown immediately following Figure 5. Less than half the code space available on the 8XC751 is used, leaving room for extra features such as parity checking and more complete keyboard control state mapping using additional look-up tables.

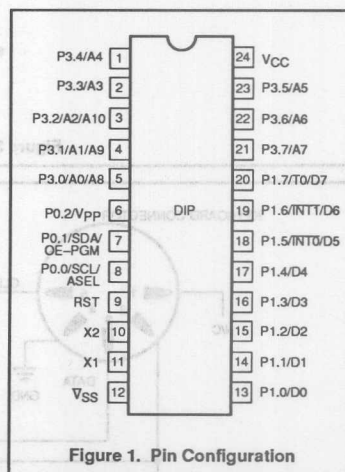
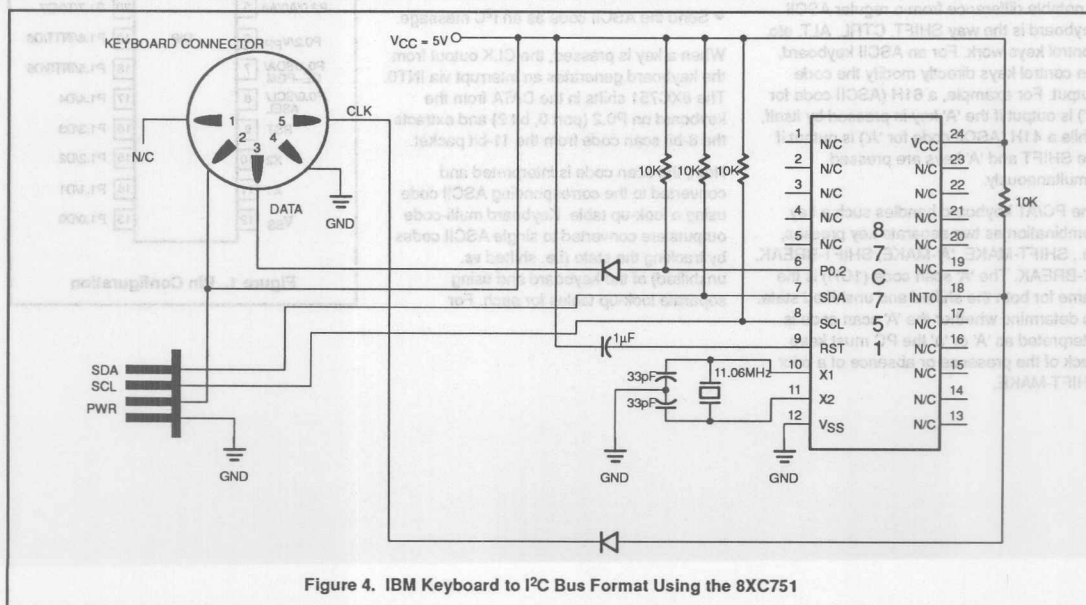
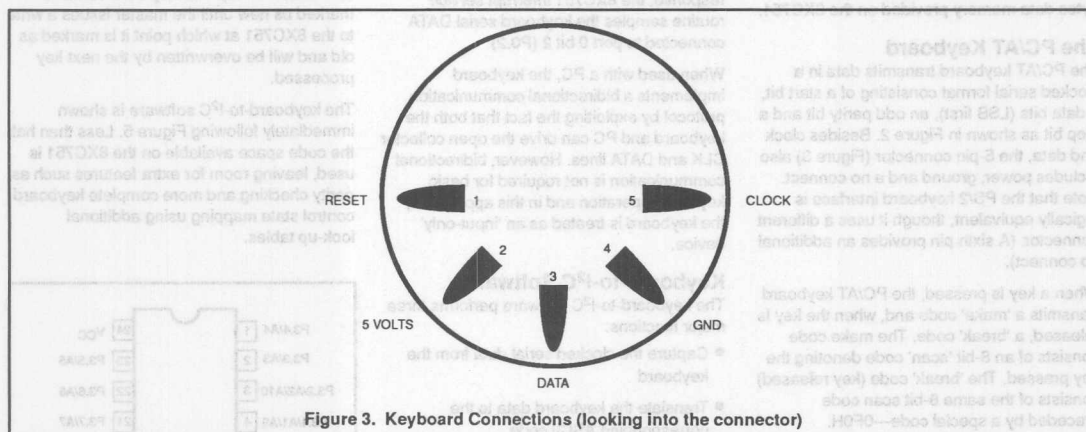
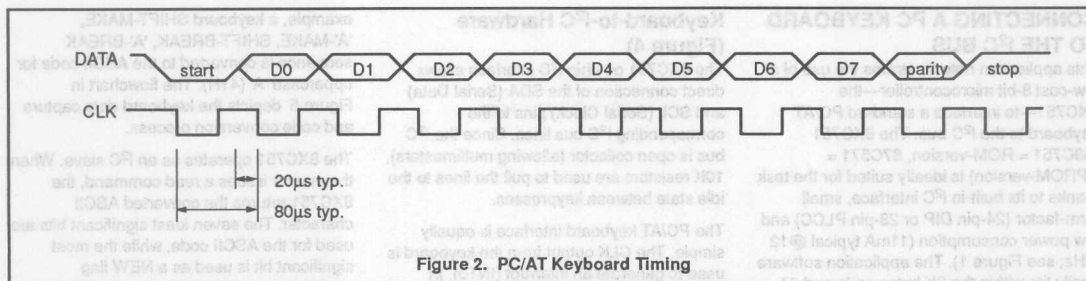
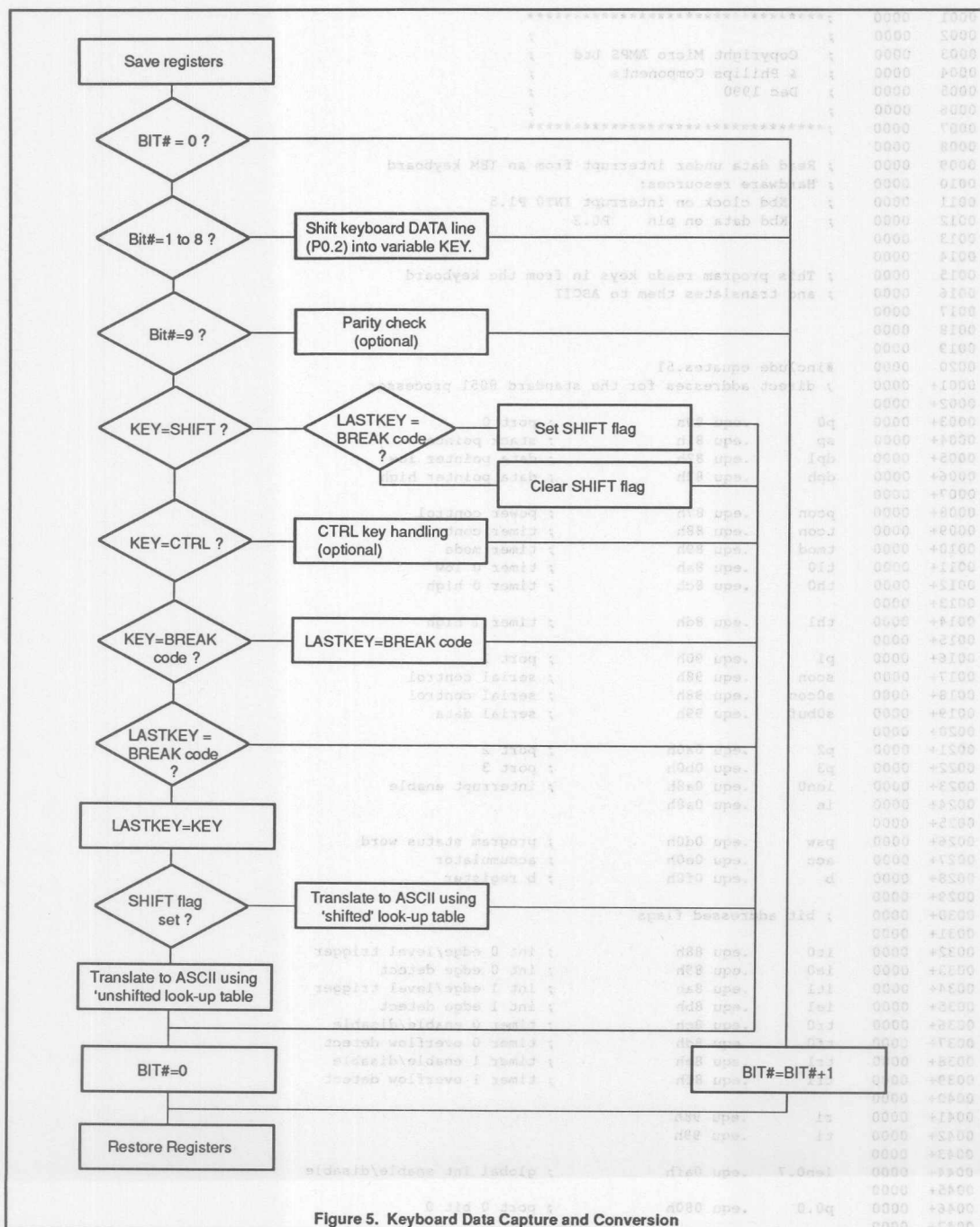
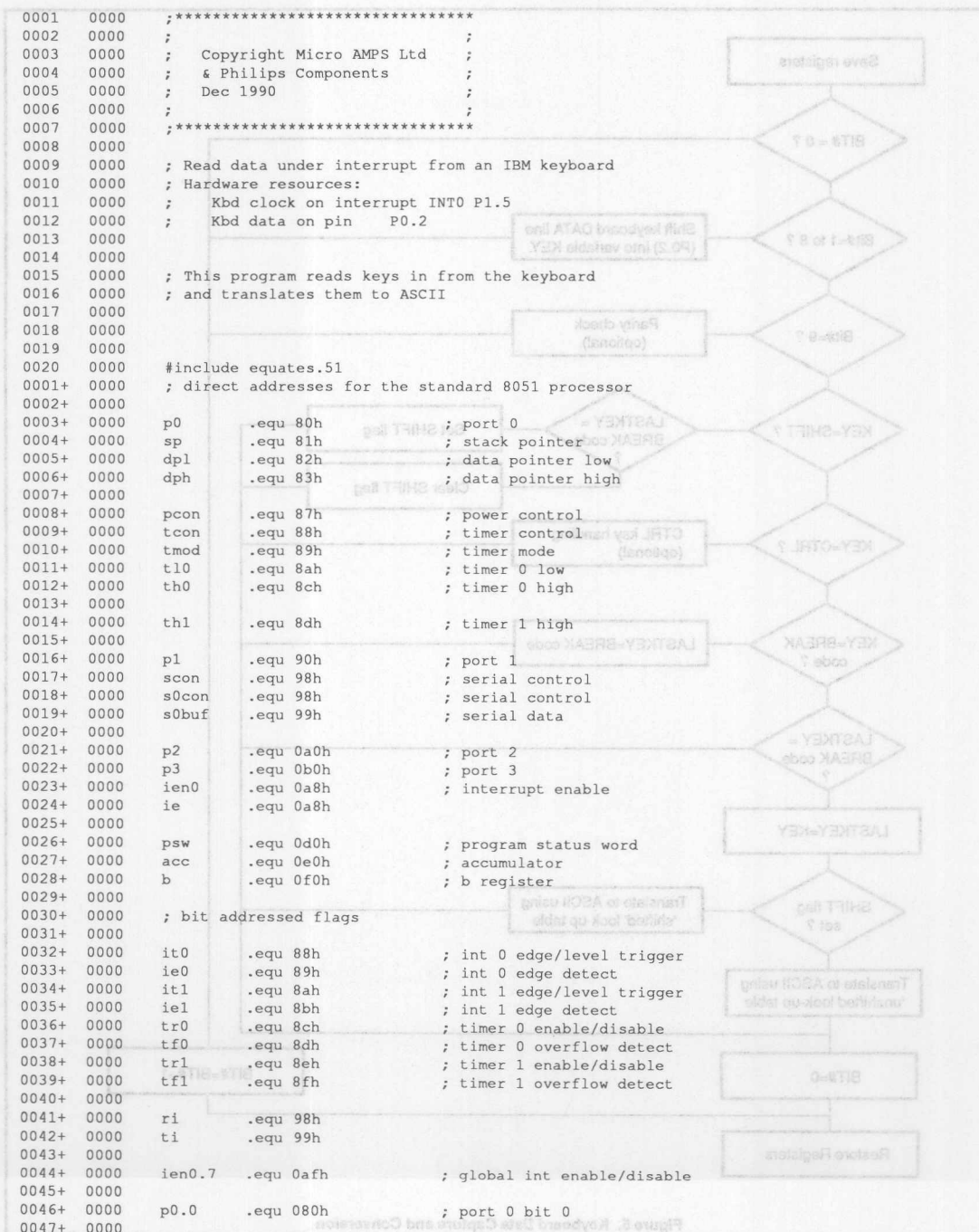


Figure 1. Pin Configuration

Connecting a PC keyboard to the I²C-bus AN434



Connecting a PC keyboard to the I²C-bus AN434

Connecting a PC keyboard to the I²C-bus AN434

Connecting a PC keyboard to the I²C-bus AN434

```

0048+ 0000 b.0 .equ 0f0h ; b reg bits
0049+ 0000 b.1 .equ 0f1h
0050+ 0000 b.2 .equ 0f2h
0051+ 0000 b.3 .equ 0f3h
0052+ 0000 b.4 .equ 0f4h
0053+ 0000 b.5 .equ 0f5h
0054+ 0000 b.6 .equ 0f6h
0055+ 0000 b.7 .equ 0f7h
0056+ 0000
0057+ 0000 a.0 .equ 0e0h ; accumulator bits
0058+ 0000 a.1 .equ 0e1h
0059+ 0000 a.2 .equ 0e2h
0060+ 0000 a.3 .equ 0e3h
0061+ 0000 a.4 .equ 0e4h
0062+ 0000 a.5 .equ 0e5h
0063+ 0000 a.6 .equ 0e6h
0064+ 0000 a.7 .equ 0e7h
0065+ 0000
0066+ 0000 rth .equ 8dh ; timer 0 reload high
0067+ 0000 rtl .equ 8bh ; timer 0 reload low
0068+ 0000
0021 0000 #include kbd.h
0001+ 0000 #define reg .equ
0002+ 0000
0003+ 0000 ;
0004+ 0000 ; 8xc751 special register set
0005+ 0000 ;
0006+ 0000 ; 751 I2C byte registers
0007+ 0000
0008+ 0000 I2CON .equ 098h ; I2C control
0009+ 0000 I2CFG .equ 0d8h ; I2C configuration
0010+ 0000 I2DAT .equ 099h ; I2C data
0011+ 0000 I2STA .equ 0f8h ; I2C status
0012+ 0000
0013+ 0000 IE .equ 0a8h ; interrupt enable
0014+ 0000
0015+ 0000 TCON .equ 088h ; timer/counter control
0016+ 0000
0017+ 0000 TL .equ 08ah ; timer 0 low
0018+ 0000 TH .equ 08ch ; timer 0 high
0019+ 0000 RTL .equ 08bh ; timer reload low
0020+ 0000 RTH .equ 08dh ; timer reload high
0021+ 0000
0022+ 0000 ; 751 I2C bit registers
0023+ 0000
0024+ 0000 ;I2CNFG
0025+ 0000
0026+ 0000 SLAVEN .equ 0dfh
0027+ 0000 MASTRQ .equ 0deh
0028+ 0000 TIRUN .equ 0dch
0029+ 0000 CT1 .equ 0d9h
0030+ 0000 CT0 .equ 0d8h
0031+ 0000 CLRTI .equ 0ddh
0032+ 0000
0033+ 0000 RDAT .equ 09fh
0034+ 0000 ATN .equ 09eh
0035+ 0000 DRDY .equ 09dh
0036+ 0000 ARL .equ 09ch
0037+ 0000 STR .equ 09bh
0038+ 0000 STP .equ 09ah
0039+ 0000 MASTER .equ 099h
0040+ 0000
0041+ 0000 ; I2CON
0042+ 0000
0043+ 0000 CXA .equ 09fh
0044+ 0000 IDLE .equ 09eh
0045+ 0000 CDR .equ 09dh

```

Connecting a PC keyboard to the I²C-bus AN434

```

0046+ 0000 CARL      .equ 09ch
0047+ 0000 CSTR      .equ 09bh
0048+ 0000 CSTP      .equ 09ah
0049+ 0000 XSTR      .equ 099h
0050+ 0000 XSTP      .equ 098h
0051+ 0000
0052+ 0000 ;I2STA
0053+ 0000
0054+ 0000 XDATA      .equ 0fdh
0055+ 0000 XACTV      .equ 0fch
0056+ 0000 MAKSTR     .equ 0fbh
0057+ 0000 MAKSTP     .equ 0fah
0058+ 0000
0059+ 0000 ; IE bit registers
0060+ 0000
0061+ 0000 EA          .equ 0afh ; clr to disable all interrupts
0062+ 0000 EI2         .equ 0ach ; set to enable iic interrupt
0063+ 0000 ETI         .equ 0abh ; set to enable timer 1 overflow interrupt
0064+ 0000 EX1         .equ 0aah ; set to enable ext int 1
0065+ 0000 ET0         .equ 0a9h ; set to enable timer 0 overflow interrupt
0066+ 0000 EX0         .equ 0a8h ; set to enable ext int 0
0067+ 0000
0068+ 0000 ; Value definitions.
0069+ 0000
0070+ 0000 CTVAL       .equ 02h ;CT1, CT0 bit values for I2C.
0071+ 0000
0072+ 0000
0073+ 0000 ; Masks for I2CFG bits.
0074+ 0000
0075+ 0000 BTIR        .equ 10h ; mask for TIRUN bit.
0076+ 0000 BMRQ        .equ 40h ; mask for MASTRO bit.
0077+ 0000
0078+ 0000
0079+ 0000 ; Masks for I2CON bits.
0080+ 0000
0081+ 0000 BCXA        .equ 80h ; mask for CXA bit.
0082+ 0000 BIDL        .equ 40h ; mask for IDLE bit.
0083+ 0000 BCDR        .equ 20h ; mask for CDR bit.
0084+ 0000 BCARL       .equ 10h ; mask for CARL bit.
0085+ 0000 BCSTR       .equ 08h ; mask for CSTR bit.
0086+ 0000 BCSTP       .equ 04h ; mask for CSTP bit.
0087+ 0000 BXSTR       .equ 02h ; mask for XSTR bit.
0088+ 0000 BXSTP       .equ 01h ; mask for XSTP bit.
0089+ 0000 ;
0090+ 0000
0091+ 0000
0092+ 0000 SCL         .equ p0.0 ; port bit for I2C serial clock line.
0093+ 0000 SDA         .equ p0.1 ; port bit for I2C serial data line.
0094+ 0000
0022 0000
0023 0000 IICADD       .equ 088h ; our I2C slave address
0024 0000 MAXBYTES     .equ 1 ; max bytes to rcv or trans
0025 0000
0026 0000 rcvdat       .equ 04h ; I2C received data buffer
0027 0000 xmtdat       .equ 06h ; I2C transmitter buffer
0028 0000
0029 0000 STACK        .equ 08h
0030 0000
0031 0000 flags         .equ 020h ; byte used as flags
0032 0000 noack         .equ (flags-20h) ; I2C flags.0, ..., 2, etc.
0033 0000 recvd         .equ (flags-20h)+1 ;
0034 0000 sent_busy     .equ (flags-20h)+2 ;
0035 0000 i2c_busy        .equ (flags-20h)+3 ;
0036 0000
0037 0000 Cntrl         .equ (flags-20h)+8 ; control key flag
0038 0000 Shift        .equ (flags-20h)+9 ; shift key flag
0039 0000

```


Connecting a PC keyboard to the I²C-bus AN434

```

0040 0000 bitcnt .equ flags+2
0041 0000 bytecnt .equ flags+3
0042 0000
0043 0000 adrrcvd .equ flags+4
0044 0000 rwflag .equ (adrrcvd-20h)*8 ; adrrcvd.0
0045 0000
0046 0000 tick .equ 025h ; count 10mS ticks to give 1sec tick
0047 0000 i2ctime .equ 027h ; I2C timeout - used on slow I2C bus
0048 0000
0049 0000
0050 0000 NBits .equ 29h ; # bits read so far
0051 0000 NBytes .equ NBits+1 ; # bytes in buffer
0052 0000 lastkey .equ NBytes+1 ; last key was?
0053 0000 keytemp .equ lastkey+1 ; used to build the key bit by bit
0054 0000 keybuff .equ keytemp+1 ; store the chars here
0055 0000
0056 0000
0057 0000 INMAX .equ 8 ; size of keyboard buffer
0058 0000 KEYCLK .equ p1.5 ; keyboard clock signal on ext int 0
0059 0000 KEYDAT .equ 82h ; keyboard data line
0060 0000
0061 0000 EDGEINT .equ 08ah
0062 0000
0063 0000 ; reset and interrupt vectors.
0064 0000 ;
0065 0000 .org 0 ; reset vector
0066 0000 01 50 ajmp start
0067 0002
0068 0003 .org 0003h ; external interrupt 0
0069 0003 01 B5 ajmp kbd
0070 0005
0071 000B .org 00bh ; counter/timer 0
0072 000B 21 EA ajmp badint
0073 000D
0074 0013 .org 013h ; external interrupt 1
0075 0013 21 EA ajmp badint
0076 0015
0077 001B .org 01bh ; timer 1 - I2C timeout
0078 001B 21 DA ajmp timer1
0079 001D
0080 0023 .org 023h ; I2C interrupt
0081 0023 21 38 ajmp i2cint
0082 0025
0083 0025
0084 0048 .org 48h
0085 0048
0086 0048 done:
0087 0048 01 48 ajmp $ ; main routine waiting for key presses
0088 004A
0089 0050 .org 50h
0090 0050
0091 0050 start:
0092 0050 78 FF mov r0,#0ffh ; power supply settling time
0093 0052 79 FF mov r1,#0ffh
0094 0054 7A 04 mov r2,#04h
0095 0056
0096 0056 D8 FE dly1:djnz r0,$
0097 0058 D9 FC djnz r1,dly1
0098 005A DA FA djnz r2,dly1
0099 005C
0100 005C reset:
0101 005C 75 80 FF mov p0,#0ffh
0102 005F 75 90 FF mov p1,#0ffh
0103 0062 75 B0 FF mov p3,#0ffh
0104 0065
0105 0065 75 81 08 mov sp,#STACK ; initialize stack pointer
0106 0068 D2 8A setb EDGEINT ; make ext int 0 edge activated

```

```

0107 006A                                ; send 13 up.         0000 0400
0108 006A C2 09      clr Shift           ; clear keyboard shift flag 0000 0401
0109 006C                                ; clear the input buffers 0000 0402
0110 006C 78 29      mov r0,#NBits       ; clear the input buffers 0000 0403
0111 006E 79 10      mov r1,#10h         ; send 10h to 0.         0000 0404
0112 0070 75 E0 00    mov acc,#0         ; count 10h ticks to give 0. 0000 0405
0113 0073                                ; do the clearing 0.         0000 0406
0114 0073 F6          clr:mov @r0,a      ; do the clearing 0.         0000 0407
0115 0074 08          inc r0              ; do the clearing 0.         0000 0408
0116 0075 D9 FC          djnz r1,clr:    ; do the clearing 0.         0000 0409
0117 0077                                ; do the clearing 0.         0000 0410
0118 0077 ;          mov xmdat,#'.'       ; transmit buffer filled with 0000 0411
0119 0077 ;          mov xmdat+1,#0ffh    ; $ff when empty.         0000 0412
0120 0077                                ; used to hold the key bit 0. 0000 0413
0121 0077 75 20 00    mov flags,#0       ; store the data here 0. 0000 0414
0122 007A 75 27 00    mov i2ctime,#0     ; store the data here 0. 0000 0415
0123 007D                                ; size of keyboard buffer 0. 0000 0416
0124 007D      restart:                  ; size of keyboard buffer 0. 0000 0417
0125 007D                                ; keyboard clock signal on the int 0. 0000 0418
0126 007D 75 D8 82    mov I2CFG,#80h+CTVAL ; enable slave functions 0. 0000 0419
0127 0080 75 98 40    mov I2CON,#BIDLE    ; place in idle state 0. 0000 0420
0128 0083                                ; enable external & IIC interrupts 0. 0000 0421
0129 0083 75 A8 91    mov ien0,#91h       ; enable external & IIC interrupts 0. 0000 0422
0130 0086                                ; reset and interrupt vectors 0. 0000 0423
0131 0086                                ; 0. 0000 0424
0132 0086 ;***** Main loop *****      ; reset vector 0. 0000 0425
0133 0086                                ; 0. 0000 0426
0134 0086      main:                    ; 0. 0000 0427
0135 0086 E5 2A      mov a,NBytes         ; if data in keybuff then 0. 0000 0428
0136 0088 60 0C      jz empty             ; copy to I2C xmt buffer 0. 0000 0429
0137 008A                                ; 0. 0000 0430
0138 008A 75 A8 00    mov ien0,#0h        ; disable all ints temporarily 0. 0000 0431
0139 008D 85 2D 06    mov xmdat,keybuff   ; 0. 0000 0432
0140 0090 75 2A 00    mov NBytes,#0       ; clear keyboard buffer full flag 0. 0000 0433
0141 0093 75 A8 91    mov ien0,#91h      ; enable external&IIC interrupts 0. 0000 0434
0142 0096                                ; 0. 0000 0435
0143 0096      empty:                  ; 0. 0000 0436
0144 0096                                ; 0. 0000 0437
0145 0096 30 01 0A    jnb recvd,notread    ; recvd flag tells 751 to clear 0. 0000 0438
0146 0099                                ; I2C xmt buffer when I2C master 0. 0000 0439
0147 0099 85 06 E0    mov acc,xmdat       ; reads the data from the 751 0. 0000 0440
0148 009C 44 80      orl a,#80h           ; the master writes any data 0. 0000 0441
0149 009E 85 E0 06    mov xmdat,acc       ; back which will set the MSB of 0. 0000 0442
0150 00A1                                ; the data buffer. This is reqd. 0. 0000 0443
0151 00A1                                ; to sync the two processors. 0. 0000 0444
0152 00A1 C2 01      clr recvd            ; reset I2C received flag 0. 0000 0445
0153 00A3                                ; 0. 0000 0446
0154 00A3      notread:                ; 0. 0000 0447
0155 00A3                                ; 0. 0000 0448
0156 00A3 E5 2B      mov a,lastkey        ; detect alt key for special 0. 0000 0449
0157 00A5 C3          clr c               ; for special functions 0. 0000 0450
0158 00A6 94 11      subb a,#11h          ; 0. 0000 0451
0159 00A8 70 01      jnz notalt           ; power supply 0. 0000 0452
0160 00AA                                ; 0. 0000 0453
0161 00AA 00          nop                 ; alt code goes here 0. 0000 0454
0162 00AB                                ; 0. 0000 0455
0163 00AB      notalt:                  ; 0. 0000 0456
0164 00AB E5 2A      mov a,NBytes         ; 0. 0000 0457
0165 00AD C3          clr c               ; 0. 0000 0458
0166 00AE 94 08      subb a,#INMAX        ; limit the input buffer to INMAX 0. 0000 0459
0167 00B0 40 01      jc notdone           ; if data is buffered 0. 0000 0460
0168 00B2                                ; then buffer overflow 0. 0000 0461
0169 00B2 00          nop                 ; code goes here 0. 0000 0462
0170 00B3                                ; 0. 0000 0463
0171 00B3                                ; 0. 0000 0464
0172 00B3      notdone:                ; 0. 0000 0465
0173 00B3                                ; 0. 0000 0466
0174 00B3 80 D1      sjmp main            ; go back to start 0. 0000 0467

```

Connecting a PC keyboard to the I²C-bus AN434

```

0175 00B5                                ;***** End of Main loop *****
0176 00B5                                ;***** External int 0 ISR *****;
0177 00B5                                ;
0178 00B5                                ; keyboard interrupt service routine ;
0179 00B5                                ;
0180 00B5                                ; ***** End of Shift Routine *****;
0181 00B5                                ; *****;
0182 00B5                                ; *****;
0183 00B5                                ; *****;
0184 00B5                                ; *****;
0185 00B5                                ; *****;
0186 00B5 kbd:                                ; save .equs during ISR
0187 00B5 C0 D0 push psw
0188 00B7 C0 E0 push acc
0189 00B9                                ; ***** Control State *****
0190 00B9 85 29 E0 mov acc,NBits                ; NBits=bit number next expected
0191 00BC                                ; from the keyboard
0192 00BC B4 00 02 cjne a,#0,bit1_8            ; if not bit 0 then bit 1 to 8
0193 00BF                                ; ***** End of Control State *****
0194 00BF                                ; *****;
0195 00BF                                ; *****;
0196 00BF ;***** Keyboard Bit 0 *****
0197 00BF bit0:                                ; discard bit 0 - Start bit
0198 00BF 80 70 sjmp bump
0199 00C1                                ; ***** Key Press *****
0200 00C1                                ; *****;
0201 00C1                                ; *****;
0202 00C1 ;***** Keyboard Bit 1-8 *****
0203 00C1                                ; *****;
0204 00C1 bit1_8:
0205 00C1 B4 09 00 cjne a,#9,$+3                ; CY flag is set if acc < 9
0206 00C4 50 0C jnc bit9
0207 00C6                                ; *****;
0208 00C6 A2 82 mov c,KEYDAT                    ; read data for keyboard data line
0209 00C8 E5 2C mov a,keytemp                    ; data arrives least sig bit 1st
0210 00CA 03 rr a                                ; hence old value is rotated and new
0211 00CB 92 E7 mov a.7,c                        ; bit is or'ed to the msb
0212 00CD 85 E0 2C mov keytemp,acc
0213 00D0 80 5F sjmp bump
0214 00D2                                ; *****;
0215 00D2                                ; *****;
0216 00D2 ;***** Bit 9 *****
0217 00D2                                ; *****;
0218 00D2 bit9:
0219 00D2 B4 09 02 cjne a,#9,bit10
0220 00D5                                ; *****;
0221 00D5 80 5A sjmp bump                        ; parity check code would go here
0222 00D7                                ; *****;
0223 00D7                                ; *****;
0224 00D7                                ; *****;
0225 00D7 ;***** Bit 10 *****
0226 00D7                                ; *****;
0227 00D7 ; The stop bit - Key Scan is now complete so convert to ASCII
0228 00D7                                ; *****;
0229 00D7 bit10:
0230 00D7 85 2C E0 mov acc,keytemp                ; get next key
0231 00DA                                ; *****;
0232 00DA B4 12 14 cjne a,#12h,notls            ; is it the left shift char?
0233 00DD                                ; *****;
0234 00DD                                ; *****;
0235 00DD ;***** Left Shift has Been Pressed *****
0236 00DD                                ; *****;
0237 00DD 85 2B E0 mov acc,lastkey                ; if last key was
0238 00E0 B4 F0 07 cjne a,#0f0h,makels            ; $f0 then shift is released
0239 00E3                                ; *****;
0240 00E3 C2 09 clr Shift                        ; next keys will be unshifted
0241 00E5 75 2B 12 mov lastkey,#12h                ; copy left shift key to last key

```

Connecting a PC keyboard to the I²C-bus AN434

```

0242 00E8 80 3F      sjmp tidy
0243 00EA            ;***** End of Main loop *****
0244 00EA      makels:
0245 00EA D2 09      setb Shift                ; next keys will be shifted
0246 00EC 75 2B 12   mov lastkey,#12h          ; copy left shift key to last key
0247 00EF 80 38      sjmp tidy
0248 00F1            ;***** End of Shift Routine *****
0249 00F1            ;*****
0250 00F1      notls:
0251 00F1            ;
0252 00F1            ; mov acc,keytemp          ; get next key
0253 00F1 B4 14 03   cjne a,#14h,notctrl       ; is it a control char?
0254 00F4            ; save codes during ISR
0255 00F4            ; push acc
0256 00F4            ;***** Control State *****
0257 00F4            ;*****
0258 00F4 00        nop                        ; control state goes here
0259 00F5 80 32      sjmp tidy
0260 00F7            ;***** End of Control State *****
0261 00F7            ;*****
0262 00F7      notctrl:
0263 00F7            ;***** Keyboard bit 0 *****
0264 00F7            ;
0265 00F7 B4 F0 04   cjne a,#0f0h,notbreak     ; if current key $f0 then break
0266 00FA            ;
0267 00FA            ;***** Key Break *****
0268 00FA            ;*****
0269 00FA F5 2B      mov lastkey,a              ; record break code in last key
0270 00FC 80 2B      sjmp tidy                  ; but don't store in the buffer
0271 00FE            ;
0272 00FE      notbreak:
0273 00FE            ;
0274 00FE 85 2B E0   mov acc,lastkey            ; if last key was $f0 then
0275 0101 B4 F0 05   cjne a,#0f0h,not_f0       ; ignore the next scan code
0276 0104            ;
0277 0104 75 2B 00   mov lastkey,#0             ; which is a break code
0278 0107 80 20      sjmp tidy
0279 0109            ;
0280 0109      not_f0:
0281 0109            ;
0282 0109            ;
0283 0109            ;***** Normal Key Press *****
0284 0109            ;*****
0285 0109~          #ifdef buffered
0286 0109~            ;
0287 0109~            ;***** Buffered Code *****
0288 0109~            ; partly check code would go here
0289 0109~            ; buffered code
0290 0109~            push 0                      ; r0 used as an indirect pointer
0291 0109~            ; so save it
0292 0109~            mov acc,#keybuff          ; copy data into keyboard
0293 0109~            add a,NBytes
0294 0109~            mov r0,a
0295 0109~            ;
0296 0109~            mov a,keytemp              ; get current key
0297 0109~            mov lastkey,a              ; & copy to lastkey
0298 0109~            ;
0299 0109~            push dph                    ; dp used to point to xlat tables
0300 0109~            push dpl                    ; since in ISR save dp contents
0301 0109~            ;
0302 0109~            jb Shift,shifted            ; if in unshifted state
0303 0109~            mov dptr,#unshift          ; use the unshift table
0304 0109~            sjmp skip1
0305 0109~            ;
0306 0109~      shifted:
0307 0109~            mov dptr,#shift              ; else use the shift table
0308 0109~            ;

```

Connecting a PC keyboard to the I²C-bus AN434

```

0309 0109~ skip1:
0310 0109~      movc a,@a+dptr      ; translate char in Acc to Ascii
0311 0109~
0312 0109~      pop dpl              ; restore the data pointer
0313 0109~      pop dph
0314 0109~
0315 0109~      cjne a,#0,Not0      ; if data is zero discard
0316 0109~
0317 0109~      ; sjmp NoSave      ; discard code goes here
0318 0109~
0319 0109~ Not0:
0320 0109~      mov r0,#keybuff      ; Save ascii value in buffer
0321 0109~      mov @r0,a          ; buffered keyboard entry
0322 0109~      inc NBytes
0323 0109~
0324 0109~ NoSave:
0325 0109~      pop 0              ; restore r0
0326 0109~
0327 0109~      ;*** End of Buffered Code
0328 0109~
0329 0109~ #endif
0330 0109~
0331 0109~
0332 0109~ #define unbuffered 1
0333 0109~
0334 0109~ #ifdef unbuffered
0335 0109~
0336 0109 E5 2C      mov a,keytemp      ; get current key
0337 010B F5 2B      mov lastkey,a      ; & copy to lastkey
0338 010D
0339 010D C0 83      push dph          ; dp used to point to xlat tables
0340 010F C0 82      push dpl          ; since in ISR save dp contents
0341 0111
0342 0111 20 09 05      jb Shift,shifted      ; if in unshifted state
0343 0114 90 01 EB      mov dptr,#unshift      ; use the unshift table
0344 0117 80 03      sjmp skip1
0345 0119
0346 0119      shifted:
0347 0119 90 02 6B      mov dptr,#shift      ; else use the shift table
0348 011C
0349 011C      skip1:
0350 011C 93      movc a,@a+dptr      ; translate char in Acc to Ascii
0351 011D
0352 011D D0 82      pop dpl          ; restore the data pointer
0353 011F D0 83      pop dph
0354 0121
0355 0121 B4 00 00      cjne a,#0,Not0      ; if data is zero discard
0356 0124
0357 0124      ; sjmp tidy      ; discard code goes here
0358 0124
0359 0124      Not0:
0360 0124 F5 2D      mov keybuff,a      ; store in keyboard buffer
0361 0126 75 2A 01      mov NBytes,#1      ; mark byte read
0362 0129
0363 0129      tidy:
0364 0129 75 29 00      mov NBits,#0      ; clear flags ready for next key
0365 012C 75 2C 00      mov keytemp,#0
0366 012F 80 02      sjmp intdone
0367 0131
0368 0131      ;***** End of Keyboard Translation and Save *****
0369 0131
0370 0131
0371 0131      ;***** Normal unfinished key exit *****
0372 0131
0373 0131      bump:
0374 0131 05 29      inc NBits          ; inc number of bits read so far
0375 0133

```


Connecting a PC keyboard to the I²C-bus AN434

```

0376 0133      intdone:
0377 0133 D0 E0      pop acc
0378 0135 D0 D0      pop psw
0379 0137 32        reti
0380 0138
0381 0138      ;***** End of Ext Int 0 ISR *****
0382 0138
0383 0138
0384 0138
0385 0138
0386 0138      ;***** I2C CODE SLAVE *****
0387 0138
0388 0138      i2cint:      ; I2C interrupt entry point
0389 0138 D2 03      setb i2c_busy      ; semaphore on xmldata buffer
0390 013A
0391 013A C0 D0      push psw      ; save registers used in ISR
0392 013C C0 E0      push acc
0393 013E C0 00      push 0
0394 0140
0395 0140 C2 AC      clr EI2      ; make I2C ISR interruptable
0396 0142 31 E9      acall clrint      ; execute a reti
0397 0144
0398 0144      slave:
0399 0144 75 27 03      mov i2ctime,#3      ; set up I2C timeout watchdog 30 ms
0400 0147
0401 0147 75 98 9C      mov I2CON,#BCARL+BCSTP+BCSTR+BCXA
0402 014A      ; clear start status
0403 014A
0404 014A 30 9E FD      jnb ATN,$      ; wait for next data bit
0405 014D 75 22 07      mov bitcnt,#7
0406 0150
0407 0150 31 C9      acall rcvrb2      ; get remainder of slave address
0408 0152 F5 24      mov adrrcvd,a
0409 0154 C2 E0      clr a.0      ; mask r/w bit to check address
0410 0156 B4 88 3B      cjne a,#IICADD,goidle ; idle again if not for us
0411 0159
0412 0159 20 20 1F      jb rwflag,read      ; test for read or write
0413 015C
0414 015C
0415 015C
0416 015C      ;***** I2C Receive Code *****
0417 015C
0418 015C 78 04      mov r0,#rcvdat      ; r0 points to data buffer
0419 015E 75 23 01      mov bytecnt,#MAXBYTES
0420 0161
0421 0161      rcvloop:
0422 0161 31 BF      acall sendack      ; acknowledge the address
0423 0163 31 C6      acall rcvbyte      ; wait for the next data byte
0424 0165 30 9D 0F      jnb DRDY,exitwr      ; end of frame
0425 0168 F6      mov @r0,a
0426 0169 08      inc r0
0427 016A D5 23 F4      djnz bytecnt,rcvloop
0428 016D
0429 016D      ; no more room
0430 016D
0431 016D 31 BF      acall sendack      ; ack last byte
0432 016F 31 C6      acall rcvbyte      ; get but discard next one
0433 0171 75 99 80      mov I2DAT,#80h      ; send neg ack
0434 0174 30 9E FD      jnb ATN,$      ; wait till gone
0435 0177      exitwr:
0436 0177
0437 0177 D2 01      setb rcvrd
0438 0179 80 13      sjmp msgend
0439 017B
0440 017B      ;***** End of Receive Routine *****
0441 017B
0442 017B

```

Connecting a PC keyboard to the I²C-bus AN434

```

0443 017B ;***** I2C Transmit Code *****
0444 017B
0445 017B read:
0446 017B 78 06 mov r0,#xmtdat ; r0 points to data buffer
0447 017D 75 23 01 mov bytecnt,#MAXBYTES
0448 0180 31 BF acall sendack ; acknowledge address
0449 0182
0450 0182 txloop:
0451 0182 E6 mov a,@r0 ; get next data byte
0452 0183 08 inc r0 ; bump buffer pointer
0453 0184 31 A7 acall xmitbyte ; transmit the byte to the I2C
0454 0186 20 00 03 jb noack,exitrd ; if not acknowledged then exit
0455 0189 D5 23 F6 djnz bytecnt,txloop
0456 018C
0457 018C 80 00 exitrd: sjmp msgend
0458 018E
0459 018E ;***** End of I2C transmit *****
0460 018E
0461 018E
0462 018E ;***** Repeated start state *****
0463 018E msgend:
0464 018E 30 9E FD jnb ATN,$ ; wait for stop or repeated start
0465 0191 20 9B B0 jb STR,slave ; if repeat start do again
0466 0194
0467 0194 ; stop so enter idle mode
0468 0194
0469 0194 goidle:
0470 0194 75 27 00 mov i2ctime,#0 ; stop I2C timeout
0471 0197 75 98 F4 mov I2CON,#BCSTP+BCXA+BCDR+BCARL+BDLE
0472 019A
0473 019A D0 00 pop 0 ; restore state before I2C ISR
0474 019C D0 E0 pop acc
0475 019E D0 D0 pop psw
0476 01A0
0477 01A0 D2 AC setb EI2
0478 01A2 D2 02 setb sent_flag ; flag to say data has been sent
0479 01A4 C2 03 clr i2c_busy ; flag denotes exiting I2C routine
0480 01A6
0481 01A6 22 ret
0482 01A7
0483 01A7
0484 01A7 ;***** General I2C routines *****
0485 01A7
0486 01A7 xmitbyte: ; transmit data in acc to I2C
0487 01A7 75 22 08 mov bitcnt,#8
0488 01AA
0489 01AA xmitbit:
0490 01AA F5 99 mov I2DAT,a
0491 01AC 23 rl a
0492 01AD 30 9E FD jnb ATN,$
0493 01B0 D5 22 F7 djnz bitcnt,xmitbit
0494 01B3 75 98 A0 mov I2CON,#BCDR+BCXA ; switch to rcv mode
0495 01B6 30 9E FD jnb ATN,$ ; wait for ack
0496 01B9 85 99 20 mov flags,I2DAT ; save ack bit
0497 01BC 22 ret
0498 01BD
0499 01BD
0500 01BD
0501 01BD rdack: ; receives data byte then sends ack
0502 01BD 31 C6 acall rcvbyte ; I2C receive, data returned in acc
0503 01BF
0504 01BF sendack:
0505 01BF 75 99 00 mov I2DAT,#0 ; I2C ack = data low and clock high
0506 01C2 30 9E FD jnb ATN,$
0507 01C5 22 ret
0508 01C6
0509 01C6 rcvbyte: ; I2C receive, data returned in acc

```

Connecting a PC keyboard to the I²C-bus AN434

```

0510 01C6 75 22 08    mov bitcnt,#8
0511 01C9
0512 01C9 E4          recvb2: clr a
0513 01CA              rllnd 22 08 00 00 00 00 00 00
0514 01CA 45 99          rbit:  orl a,I2DAT
0515 01CC 23            rl a
0516 01CD 30 9E FD      jnb ATN,$
0517 01D0 30 9D 06      jnb DRDY,rbex
0518 01D3 D5 22 F4      djnz bitcnt,rbit
0519 01D6
0520 01D6 A2 9F 00      mov c,RDAT
0521 01D8 33            rlc a
0522 01D9
0523 01D9              rbex:
0524 01D9 22            ret
0525 01DA
0526 01DA              ; IIC timer interrupt service
0527 01DA timerI:
0528 01DA 75 A8 00      mov ien0,#0
0529 01DD D2 DD          setb CLRTI
0530 01DF fixup:
0531 01DF 75 D8 00      mov I2CFG,#0
0532 01E2 75 98 BC      mov I2CON,#BCXA+BCDR+BCARL+BCSTR+BCSTP
0533 01E5
0534 01E5 31 E9          acall clrint
0535 01E7 01 5C          ajmp reset
0536 01E9
0537 01E9
0538 01E9              ;***** call here to make code interruptible*****
0539 01E9
0540 01E9 clrint:
0541 01E9 32            reti
0542 01EA
0543 01EA              ;***** unused interrupts are vectored to here *****
0544 01EA badint:
0545 01EA 32            reti
0546 01EB
0547 01EB #include attable.h
0001+ 01EB unshift
0002+ 01EB 00           .byte 0 ; scan code
0003+ 01EC 00           .byte 0 ; 1 - f9
0004+ 01ED 00           .byte 0 ; 2 - f7
0005+ 01EE 00           .byte 0 ; 3 - f5
0006+ 01EF 00           .byte 0 ; 4 - f3
0007+ 01F0 00           .byte 0 ; 5 - f1
0008+ 01F1 00           .byte 0 ; 6 - f2
0009+ 01F2 00           .byte 0 ; 7 - f2
0010+ 01F3 00           .byte 0 ; 8 -
0011+ 01F4 00           .byte 0 ; 9 - f10
0012+ 01F5 00           .byte 0 ; a - f8
0013+ 01F6 00           .byte 0 ; b - f6
0014+ 01F7 00           .byte 0 ; c - f4
0015+ 01F8 09           .byte 09h ; d - tab
0016+ 01F9 60           .byte ' ' ; e - '
0017+ 01FA 00           .byte 0 ; f -
0018+ 01FB
0019+ 01FB 00           .byte 0 ; 10
0020+ 01FC 00           .byte 0 ; 11 - left shift
0021+ 01FD 00           .byte 0 ; 12 - right shift
0022+ 01FE 00           .byte 0 ; 13 - right shift
0023+ 01FF 00           .byte 0 ; 14
0024+ 0200 71           .byte 'q' ; 15
0025+ 0201 31           .byte '1' ; 16
0026+ 0202 00           .byte 0 ; 17
0027+ 0203 00           .byte 0 ; 18
0028+ 0204 00           .byte 0 ; 19
0029+ 0205 7A           .byte 'z' ; 1a

```

Connecting a PC keyboard to the I²C-bus AN434

0030+	0206	73	.byte 's' ; 1b	0030+ 0206 73 .byte 's' ; 1b
0031+	0207	61	.byte 'a' ; 1c	0031+ 0207 61 .byte 'a' ; 1c
0032+	0208	77	.byte 'w' ; 1d	0032+ 0208 77 .byte 'w' ; 1d
0033+	0209	32	.byte '2' ; 1e	0033+ 0209 32 .byte '2' ; 1e
0034+	020A	00	.byte 0 ; 1f	0034+ 020A 00 .byte 0 ; 1f
0035+	020B			0035+ 020B .byte 0 ; 20
0036+	020B	00	.byte 0 ; 20	0036+ 020B 00 .byte 0 ; 20
0037+	020C	63	.byte 'c' ; 21	0037+ 020C 63 .byte 'c' ; 21
0038+	020D	78	.byte 'x' ; 22	0038+ 020D 78 .byte 'x' ; 22
0039+	020E	64	.byte 'd' ; 23	0039+ 020E 64 .byte 'd' ; 23
0040+	020F	65	.byte 'e' ; 24	0040+ 020F 65 .byte 'e' ; 24
0041+	0210	34	.byte '4' ; 25	0041+ 0210 34 .byte '4' ; 25
0042+	0211	33	.byte '3' ; 26	0042+ 0211 33 .byte '3' ; 26
0043+	0212	00	.byte 0 ; 27	0043+ 0212 00 .byte 0 ; 27
0044+	0213	00	.byte 0 ; 28	0044+ 0213 00 .byte 0 ; 28
0045+	0214	20	.byte ' ' ; 29	0045+ 0214 20 .byte ' ' ; 29
0046+	0215	76	.byte 'v' ; 2a	0046+ 0215 76 .byte 'v' ; 2a
0047+	0216	66	.byte 'f' ; 2b	0047+ 0216 66 .byte 'f' ; 2b
0048+	0217	74	.byte 't' ; 2c	0048+ 0217 74 .byte 't' ; 2c
0049+	0218	72	.byte 'r' ; 2d	0049+ 0218 72 .byte 'r' ; 2d
0050+	0219	35	.byte '5' ; 2e	0050+ 0219 35 .byte '5' ; 2e
0051+	021A	00	.byte 0 ; 2f	0051+ 021A 00 .byte 0 ; 2f
0052+	021B			0052+ 021B .byte 0 ; 30
0053+	021B	00	.byte 0 ; 30	0053+ 021B 00 .byte 0 ; 30
0054+	021C	6E	.byte 'n' ; 31	0054+ 021C 6E .byte 'n' ; 31
0055+	021D	62	.byte 'b' ; 32	0055+ 021D 62 .byte 'b' ; 32
0056+	021E	68	.byte 'h' ; 33	0056+ 021E 68 .byte 'h' ; 33
0057+	021F	67	.byte 'g' ; 34	0057+ 021F 67 .byte 'g' ; 34
0058+	0220	79	.byte 'y' ; 35	0058+ 0220 79 .byte 'y' ; 35
0059+	0221	36	.byte '6' ; 36	0059+ 0221 36 .byte '6' ; 36
0060+	0222	00	.byte 0 ; 37	0060+ 0222 00 .byte 0 ; 37
0061+	0223	00	.byte 0 ; 38	0061+ 0223 00 .byte 0 ; 38
0062+	0224	00	.byte 0 ; 39	0062+ 0224 00 .byte 0 ; 39
0063+	0225	6D	.byte 'm' ; 3a	0063+ 0225 6D .byte 'm' ; 3a
0064+	0226	6A	.byte 'j' ; 3b	0064+ 0226 6A .byte 'j' ; 3b
0065+	0227	75	.byte 'u' ; 3c	0065+ 0227 75 .byte 'u' ; 3c
0066+	0228	37	.byte '7' ; 3d	0066+ 0228 37 .byte '7' ; 3d
0067+	0229	38	.byte '8' ; 3e	0067+ 0229 38 .byte '8' ; 3e
0068+	022A	00	.byte 0 ; 3f	0068+ 022A 00 .byte 0 ; 3f
0069+	022B			0069+ 022B .byte 0 ; 40
0070+	022B	00	.byte 0 ; 40	0070+ 022B 00 .byte 0 ; 40
0071+	022C	2C	.byte ',' ; 41	0071+ 022C 2C .byte ',' ; 41
0072+	022D	6B	.byte 'k' ; 42	0072+ 022D 6B .byte 'k' ; 42
0073+	022E	69	.byte 'i' ; 43	0073+ 022E 69 .byte 'i' ; 43
0074+	022F	6F	.byte 'o' ; 44	0074+ 022F 6F .byte 'o' ; 44
0075+	0230	30	.byte '0' ; 45	0075+ 0230 30 .byte '0' ; 45
0076+	0231	39	.byte '9' ; 46	0076+ 0231 39 .byte '9' ; 46
0077+	0232	00	.byte 0 ; 47	0077+ 0232 00 .byte 0 ; 47
0078+	0233	00	.byte 0 ; 48	0078+ 0233 00 .byte 0 ; 48
0079+	0234	2E	.byte ',' ; 49	0079+ 0234 2E .byte ',' ; 49
0080+	0235	2F	.byte '/' ; 4a	0080+ 0235 2F .byte '/' ; 4a
0081+	0236	6C	.byte 'l' ; 4b	0081+ 0236 6C .byte 'l' ; 4b
0082+	0237	00	.byte ' ' ; 4c	0082+ 0237 00 .byte ' ' ; 4c
0083+	0238	70	.byte 'p' ; 4d	0083+ 0238 70 .byte 'p' ; 4d
0084+	0239	2D	.byte '-' ; 4e	0084+ 0239 2D .byte '-' ; 4e
0085+	023A	00	.byte 0 ; 4f	0085+ 023A 00 .byte 0 ; 4f
0086+	023B			0086+ 023B .byte 0 ; 50
0087+	023B	00	.byte 0 ; 50	0087+ 023B 00 .byte 0 ; 50
0088+	023C	2C	.byte ',' ; 51	0088+ 023C 2C .byte ',' ; 51
0089+	023D	00	.byte 0 ; 52	0089+ 023D 00 .byte 0 ; 52
0090+	023E	00	.byte 0 ; 53	0090+ 023E 00 .byte 0 ; 53
0091+	023F	5B	.byte '[' ; 54	0091+ 023F 5B .byte '[' ; 54
0092+	0240	3D	.byte '=' ; 55	0092+ 0240 3D .byte '=' ; 55
0093+	0241	00	.byte 0 ; 56	0093+ 0241 00 .byte 0 ; 56
0094+	0242	00	.byte 0 ; 57	0094+ 0242 00 .byte 0 ; 57
0095+	0243	00	.byte 0 ; 58	0095+ 0243 00 .byte 0 ; 58
0096+	0244	00	.byte 0 ; 59	0096+ 0244 00 .byte 0 ; 59

Connecting a PC keyboard to the I²C-bus AN434

```

0097+ 0245 0D      .byte 13      ; 5a
0098+ 0246 5D      .byte ']'      ; 5b
0099+ 0247 00      .byte 0       ; 5c
0100+ 0248 5C      .byte 92      ; 5d
0101+ 0249 00      .byte 0       ; 5e
0102+ 024A 00      .byte 0       ; 5f
0103+ 024B
0104+ 024B 00      .byte 0       ; 60
0105+ 024C 00      .byte 0       ; 61
0106+ 024D 00      .byte 0       ; 62
0107+ 024E 00      .byte 0       ; 63
0108+ 024F 00      .byte 0       ; 64
0109+ 0250 00      .byte 0       ; 65
0110+ 0251 08      .byte 8       ; 66
0111+ 0252 00      .byte 0       ; 67
0112+ 0253 00      .byte 0       ; 68
0113+ 0254 00      .byte 0       ; 69
0114+ 0255 00      .byte 0       ; 6a
0115+ 0256 00      .byte 0       ; 6b
0116+ 0257 00      .byte 0       ; 6c
0117+ 0258 00      .byte 0       ; 6d
0118+ 0259 00      .byte 0       ; 6e
0119+ 025A 00      .byte 0       ; 6f
0120+ 025B
0121+ 025B 00      .byte 0       ; 70
0122+ 025C 7F      .byte 127    ; 71
0123+ 025D 00      .byte 0       ; 72
0124+ 025E 00      .byte 0       ; 73
0125+ 025F 00      .byte 0       ; 74
0126+ 0260 1B      .byte 27     ; 75
0127+ 0261 00      .byte 0       ; 76
0128+ 0262 00      .byte 0       ; 77
0129+ 0263 00      .byte 0       ; 78
0130+ 0264 2B      .byte '+'     ; 79
0131+ 0265 00      .byte 0       ; 7a
0132+ 0266 2D      .byte '-'     ; 7b
0133+ 0267 2A      .byte '*'     ; 7c
0134+ 0268 00      .byte 0       ; 7d
0135+ 0269 00      .byte 0       ; 7e
0136+ 026A 00      .byte 0       ; 7f
0137+ 026B
0138+ 026B      shift:      ; scan code
0139+ 026B 00      .byte 0       ; 0
0140+ 026C 00      .byte 0       ; 1 - f9
0141+ 026D 00      .byte 0       ; 2 - f7
0142+ 026E 00      .byte 0       ; 3 - f5
0143+ 026F 00      .byte 0       ; 4 - f3
0144+ 0270 00      .byte 0       ; 5 - f1
0145+ 0271 00      .byte 0       ; 6 - f2
0146+ 0272 00      .byte 0       ; 7 - f2
0147+ 0273 00      .byte 0       ; 8 -
0148+ 0274 00      .byte 0       ; 9 - f10
0149+ 0275 00      .byte 0       ; a - f8
0150+ 0276 00      .byte 0       ; b - f6
0151+ 0277 00      .byte 0       ; c - f4
0152+ 0278 00      .byte 0       ; d - tab
0153+ 0279 7E      .byte '~'     ; e - ~
0154+ 027A 00      .byte 0       ; f -
0155+ 027B
0156+ 027B 00      .byte 0       ; 10
0157+ 027C 00      .byte 0       ; 11 -
0158+ 027D 00      .byte 0       ; 12
0159+ 027E 00      .byte 0       ; 13
0160+ 027F 00      .byte 0       ; 14
0161+ 0280 51      .byte 'Q'     ; 15
0162+ 0281 21      .byte '!'     ; 16
0163+ 0282 00      .byte 0       ; 17

```


Connecting a PC keyboard to the I²C-bus AN434

```

0164+ 0283 00      .byte 0      ; 18
0165+ 0284 00      .byte 0      ; 19
0166+ 0285 5A      .byte 'Z'    ; 1a
0167+ 0286 53      .byte 'S'    ; 1b
0168+ 0287 41      .byte 'A'    ; 1c
0169+ 0288 57      .byte 'W'    ; 1d
0170+ 0289 40      .byte '@'    ; 1e
0171+ 028A 00      .byte 0      ; 1f
0172+ 028B

0173+ 028B 00      .byte 0      ; 20
0174+ 028C 43      .byte 'C'    ; 21
0175+ 028D 58      .byte 'X'    ; 22
0176+ 028E 44      .byte 'D'    ; 23
0177+ 028F 45      .byte 'E'    ; 24
0178+ 0290 24      .byte '$'    ; 25
0179+ 0291 23      .byte '#'    ; 26
0180+ 0292 00      .byte 0      ; 27
0181+ 0293 00      .byte 0      ; 28
0182+ 0294 20      .byte ' '    ; 29
0183+ 0295 56      .byte 'V'    ; 2a
0184+ 0296 46      .byte 'F'    ; 2b
0185+ 0297 54      .byte 'T'    ; 2c
0186+ 0298 52      .byte 'R'    ; 2d
0187+ 0299 25      .byte ''     ; 2e
0188+ 029A 00      .byte 0      ; 2f
0189+ 029B

0190+ 029B 00      .byte 0      ; 30
0191+ 029C 4E      .byte 'N'    ; 31
0192+ 029D 42      .byte 'B'    ; 32
0193+ 029E 48      .byte 'H'    ; 33
0194+ 029F 47      .byte 'G'    ; 34
0195+ 02A0 59      .byte 'Y'    ; 35
0196+ 02A1 5E      .byte '^'    ; 36
0197+ 02A2 00      .byte 0      ; 37
0198+ 02A3 00      .byte 0      ; 38
0199+ 02A4 00      .byte 0      ; 39
0200+ 02A5 4D      .byte 'M'    ; 3a
0201+ 02A6 4A      .byte 'J'    ; 3b
0202+ 02A7 55      .byte 'U'    ; 3c
0203+ 02A8 26      .byte '&'    ; 3d
0204+ 02A9 2A      .byte '**'   ; 3e
0205+ 02AA 00      .byte 0      ; 3f
0206+ 02AB

0207+ 02AB 00      .byte 0      ; 40
0208+ 02AC 3C      .byte '<'    ; 41
0209+ 02AD 4B      .byte 'K'    ; 42
0210+ 02AE 49      .byte 'I'    ; 43
0211+ 02AF 4F      .byte 'O'    ; 44
0212+ 02B0 29      .byte ')'    ; 45
0213+ 02B1 28      .byte '('    ; 46
0214+ 02B2 00      .byte 0      ; 47
0215+ 02B3 00      .byte 0      ; 48
0216+ 02B4 3E      .byte '>'    ; 49
0217+ 02B5 3F      .byte '?'    ; 4a
0218+ 02B6 4C      .byte 'L'    ; 4b
0219+ 02B7 3A      .byte ':'    ; 4c
0220+ 02B8 50      .byte 'P'    ; 4d
0221+ 02B9 5F      .byte '-'    ; 4e
0222+ 02BA 00      .byte 0      ; 4f
0223+ 02BB

0224+ 02BB 00      .byte 0      ; 50
0225+ 02BC 00      .byte 0      ; 51
0226+ 02BD 22      .byte '"'    ; 52
0227+ 02BE 00      .byte 0      ; 53
0228+ 02BF 7B      .byte '{'    ; 54
0229+ 02C0 2B      .byte '+'    ; 55
0230+ 02C1 00      .byte 0      ; 56

```

Connecting a PC keyboard to the I²C-bus AN434

```

0231+ 02C2 00      .byte 0      ; 57
0232+ 02C3 00      .byte 0      ; 58
0233+ 02C4 00      .byte 0      ; 59
0234+ 02C5 0D      .byte 13     ; 5a
0235+ 02C6 7D      .byte '}'    ; 5b
0236+ 02C7 00      .byte 0      ; 5c
0237+ 02C8 7C      .byte '|'    ; 5d
0238+ 02C9 00      .byte 0      ; 5e
0239+ 02CA 00      .byte 0      ; 5f
0240+ 02CB
0241+ 02CB 00      .byte 0      ; 60
0242+ 02CC 00      .byte 0      ; 61
0243+ 02CD 00      .byte 0      ; 62
0244+ 02CE 00      .byte 0      ; 63
0245+ 02CF 00      .byte 0      ; 64
0246+ 02D0 00      .byte 0      ; 65
0247+ 02D1 08      .byte 8      ; 66
0248+ 02D2 00      .byte 0      ; 67
0249+ 02D3 00      .byte 0      ; 68
0250+ 02D4 31      .byte '1'    ; 69
0251+ 02D5 00      .byte 0      ; 6a
0252+ 02D6 34      .byte '4'    ; 6b
0253+ 02D7 37      .byte '7'    ; 6c
0254+ 02D8 00      .byte 0      ; 6d
0255+ 02D9 00      .byte 0      ; 6e
0256+ 02DA 00      .byte 0      ; 6f
0257+ 02DB
0258+ 02DB 30      .byte '0'    ; 70
0259+ 02DC 2E      .byte '.'    ; 71
0260+ 02DD 32      .byte '2'    ; 72
0261+ 02DE 35      .byte '5'    ; 73
0262+ 02DF 36      .byte '6'    ; 74
0263+ 02E0 38      .byte '8'    ; 75
0264+ 02E1 1B      .byte 27     ; 76
0265+ 02E2 00      .byte 0      ; 77
0266+ 02E3 00      .byte 0      ; 78
0267+ 02E4 2B      .byte '+'    ; 79
0268+ 02E5 33      .byte '3'    ; 7a
0269+ 02E6 2D      .byte '-'    ; 7b
0270+ 02E7 00      .byte 0      ; 7c
0271+ 02E8 39      .byte '9'    ; 7d
0272+ 02E9 00      .byte 0      ; 7e
0273+ 02EA 00      .byte 0      ; 7f
0274+ 02EB
0275+ 02EB
0548 02EB
0549 02EB      .end
tasm: Number of errors = 0

```

Multimaster I²C routines for byte-oriented I²C interfaces AN435

DESCRIPTION

This application note consists of a set of drivers to allow easy use of multimaster I²C on Philips microcontrollers that have the byte oriented I²C interface. Some devices that include this version of the I²C interface are the 8XC552, 8XC562, 8XC652, 8XC654,

8XCL410, 8XCL580, and the 8XCL781. This program is used as an I²C driver which communicates with the user's main program using a simple macro language.

The source code file for this program is available for downloading from the Philips

computer bulletin board system. This system is open to all callers, operates 24 hours a day, and can be accessed with modems at 2400, 1200, and 300 baud. The telephone numbers for the BBS are: (800) 451-6644 (in the U.S. only) or (408) 991-2406.

```
$Title(I2C Byte Oriented Software Driver)
$Date(04/22/92)
;
;*****
;I2C Byte Oriented System Driver.
;Written by Joe Brandolino, FAE, Etobicoke Sales Office.
;Region 47 (Canada).
;*****
;
;DESCRIPTION:
;=====
;IIC_OS.ASM contains a complete multimaster I2C driver for the byte
;oriented Philips microcontrollers. To date, the list of byte
;oriented 80C51 derivative microcontrollers includes:
;
;      - 8XC552
;      - 8XC562
;      - 8XC652
;      - 8XC654
;      - 8XCL410
;      - 8XCL580
;      - 8XCL781
;
;IIC_OS was written for Philips customers who do not want to spend
;the many hours required to develop a complete multimaster IIC driver. This
;program is used as an IIC driver which communicates with the main program
;using a simple macro language.
;
;The comments in this listing assume that the reader has a basic knowledge of
;the 80C51 family, and is familiar with IIC basics. This program has been
;tested as thoroughly as time permitted; however, Philips cannot
;guarantee that this IIC driver is flawless in all applications.
;
;The comment text fields in this file use a consistent method of highlighting
;the various parameters of the software. All constants (EQUates), registers,
;bits and other bytes are surrounded by ' ' in the comment text. All routines,
;labels, procedures and file names are surrounded by " " in the comment text.
;Generally speaking, all 8051 mnemonics are in UPPERCASE, all variable names
;and labels are in LOWERCASE or mixed case. The terms IIC and I2C are used
;interchangeably, and both mean Inter-Integrated Circuit.
;
;---NOTE---NOTE---NOTE---NOTE---NOTE---NOTE---NOTE---NOTE---NOTE---NOTE---
;
;To incorporate this program into your main program, place it somewhere in
;your source text file by including the following text:
;
;      $include(mod552)      ;include the desired processor descriptor file
;      $include(iic_os.asm)  ;include this program
;
;Since this program has a 'CSEG AT... definition for the IIC interrupt vector,
;it is probably best to place it in your program where all the other interrupt
;vector directives reside so that assembly synchronization errors do not
```

Multimaster I²C routines for byte-oriented I²C interfaces AN435

```

; occur.
;
; You must also ensure that the data bytes used by this program do not
; conflict with those in your main program. Don't forget to initialize
; the IIC control registers and the interrupt registers, etc. For example:
;
; INIT:
;
;
; MOV     P1,#0
; MOV     P1,#0FFH
; MOV     IEN0,#10100000B
; MOV     S1CON,#ENSI_NOTSTA_NOTSTO_NOTSI_AA_CRO
; MOV     S1ADR,#Own_adrs OR general_enable
; MOV     IIC_status,#status_OK
; CLR     IIC_failure
;
;
;
; This driver uses DATA space bytes from decimal address 48 to 78 (16 of these
; bytes are for slave mode receive and transmit buffers - this space can be re-used
; if not required). Bit space addresses used are from 0 to 9 decimal. These
; addresses can be moved to any convenient location in your system. If the
; driver is used as is, then start your DATA space definitions at 'DATA_start'
; decimal (i.e. DSEG AT DATA_start) and your BIT space definitions at
; 'BITS_start' (i.e. BSEG AT BITS_start). There are no register banks used per
; se - all registers required are pushed onto the stack if used.
; To interface to this IIC Driver, the user need not understand all the details
; of the program - only the following registers must be understood:
;
; 'ICC_Command_file_adrs' - used in every command file
; 'indirect_adrs' - used only with 'indirect-' option
; 'indirect_count' - used only with 'indirect-' option
; 'single_data' - used only with 'singleD-' option
; 'Slave_in' buffer (if required) - used only in multimaster systems
; 'Slave_out' buffer (if required) - used only in multimaster systems
; 'IIC_failure' (BIT) - set if command file was kaput
; 'IIC_status' - holds final status of session
;
; Additionally, there is a command file structure (the command file is a
; list of commands that "IIC_OS" will execute) which the user must conform to.
; The list of commands includes:
;
; 'ioD_' - target DATA space for I/O transfers
; 'ioC_' - target CODE space for I/O transfers
; 'ioX_' - target XDATA space for I/O transfers
; 'immediate_' - used to output 1 byte from command file stream
; 'call' - used to call a subroutine between repeated starts
; 'indirect_' - gets I/O address and count from 'indirect_registers'
; 'singleD_' - gets 1 byte of I/O data from 'single_data'
;
; 'iicend' - last byte of a command file
; 'iicwritemask' - OR with slave address to indicate a write operation
; 'iicreadmask' - OR with slave address to indicate a read operation
;
; The command file structure is explained in detail below.
; ---NOTE---NOTE---NOTE---NOTE---NOTE---NOTE---NOTE---NOTE---NOTE---NOTE---
;
; Multimaster systems are very specific to the system design, and therefore,
; very difficult to make generic. Every multimaster system will have a
; different protocol for how many (and which) bytes to send/receive when the

```

Multimaster I²C routines for byte-oriented I²C interfaces AN435

```

;master is addressed as a Slave Receiver or Slave Transmitter. For this
;reason, this program implements the multimaster scenario very simply -
;if the micro running this program is addressed as a slave, it will read
;'SLVbytes_in' number of data bytes or write 'SLVbytes_out' number of data
;bytes (depending on what the calling master requests). The target data
;buffer in these cases are the 'Slave_in' buffer and the 'Slave_out' buffer.
;
;The user can make the size of these slave input/output buffers (and the
;corresponding equates 'SLVbytes_in' and 'SLVbytes_out') as large as
;required. The calling Master can terminate the slave session at any number
;of data bytes sent or received by providing a stop or a not acknowledge.
;
;IIC_OS, when integrated into the user's system, will require 15 DATA bytes
;(mapped anywhere in the internal DATA memory space), and one bit-addressable
;byte. About 600 bytes of code-space memory are used.
;
;The user of this program need not concern himself with the bit or byte level
;operation of the IIC hardware - this program takes care of all IIC registers,
;and checks for all collisions, arbitration lost scenarios, bus errors, etc.
;A command list consisting of a limited number of simple macro commands is
;set-up by the user, and this driver uses that list of commands to perform
;the desired IIC operations.
;
;The user loads the 'IIC_Command_File_adrs' (2 byte) register with
;the address of the sequence of IIC operations desired. Once this register
;is loaded, the "WAIT_IIC_Data", "WAIT_IIC_Xdata", or "WAIT_IIC_Code" routine
;is called, depending on which data space the command file list resides.
;
;"WAIT_IIC" starts the IIC interrupt service routine by setting the 'STA'
;(IIC start) bit. Then the "IIC_VECTOR" routine is entered after every
;significant IIC event (this occurs because of the IIC hardware in the
;microcontroller). The interrupt service routine takes care of setting the
;IIC hardware registers and checking for collisions and stepping through the
;IIC command file. "WAIT_IIC" also does a timeout feature for the IIC system.
;
;The IIC operations to be performed are stored sequentially starting at the
;address specified by the 'IIC_Command_File_adrs' and in the memory space
;designated by 'Command?adrs?space' (the later register is loaded with the
;appropriate memory space code through the call to the "WAIT_IIC_xxxx"
;routine). IIC operations include:
;
;1) sending or receiving any number of bytes from 1 to 255
;   into any valid address space
;2) repeated start automatically performed so multiple
;   slaves can be communicated with in one call
;3) call subroutines between repeated start conditions directly
;   from the IIC command file list (i.e. transparent to the
;   calling routine).
;
;The IIC Command File must be constructed so that it conforms to the IIC
;driver system format. This format is very simple and is outlined later.
;The IIC Command File is built from 1 to any number of blocks. Each block
;is from 2 to 8 bytes long, depending on what functions must be performed.
;There are only eight types of blocks, indicated by the options in the format
;below and briefly explained here:
;
;option 1 FUNCTION: send/receive bytes to/from slave from/to any memory space
;# BYTES IN THIS COMMAND FILE BLOCK:
;NUMBER OF BYTES TO SEND/RECEIVE: get number from command file
;ADDRESS FOR DATA DERIVED FROM: command file
;OTHER FUNCTIONS: none

```


Multimaster I²C routines for byte-oriented I²C interfaces AN435

```

; COMMENTS: Option 1 is useful for sending or receiving any specified
;           number of data bytes to/from the specified slave. Every
;           required piece of information is stored in the command
;           file - that is, the address of the slave + read/write bit,
;           the number of bytes to send or receive, and the address
;           to send from or receive to. Recall that the address space
;           is always specified in the command file.
;
;option 2 FUNCTION:      send one byte to slave from command file
; # BYTES IN THIS COMMAND FILE BLOCK: 3
; NUMBER OF BYTES TO SEND/RECEIVE: 1
; ADDRESS FOR DATA DERIVED FROM: data read directly from command file
; OTHER FUNCTIONS:      none
; COMMENTS: Option 2 is used to send exactly one byte to an addressed
;           slave. The byte is fixed and is stored in the command
;           file itself; this method reduces command file bytes since
;           no specification for data address or number of data bytes
;           is necessary. Option 2 provides a simple means of setting
;           the sub-address in IIC memory devices.
;
;option 3 FUNCTION:      send/receive bytes to/from slave from/to any memory space
; # BYTES IN THIS COMMAND FILE BLOCK: 2
; NUMBER OF BYTES TO SEND/RECEIVE: get number from 'indirect_count'
; ADDRESS FOR DATA DERIVED FROM: get address from 'indirect_address'
; OTHER FUNCTIONS:      none
; COMMENTS: Option 3 assumes that the calling program has set-up the
;           'indirect_count' register with the number of bytes to be
;           sent or received, and the 'indirect_address' with the
;           address of the bytes to be sent or received. The data
;           space targeted is specified in the command file as
;           usual.
;
;option 4 FUNCTION:      send/receive bytes to/from slave from/to any memory space
; # BYTES IN THIS COMMAND FILE BLOCK: 7
; NUMBER OF BYTES TO SEND/RECEIVE: get number from command file
; ADDRESS FOR DATA DERIVED FROM: get address from command file
; OTHER FUNCTIONS:      CALL subroutine listed in command file
; COMMENTS: Option 4 is identical to Option 1, except that a
;           subroutine (whose address is specified in the command
;           file) is called after the data transfer is complete.
;
;option 5 FUNCTION:      send one byte to slave from command file
; # BYTES IN THIS COMMAND FILE BLOCK: 5
; NUMBER OF BYTES TO SEND/RECEIVE: 1
; ADDRESS FOR DATA DERIVED FROM: data read directly from command file
; OTHER FUNCTIONS:      CALL subroutine listed in command file
; COMMENTS: Option 5 is identical to Option 2, except that a
;           subroutine (whose address is specified in the command
;           file) is called after the data transfer is complete.
;
;option 6 FUNCTION:      send/receive bytes to/from slave from/to any memory space
; # BYTES IN THIS COMMAND FILE BLOCK: 4
; NUMBER OF BYTES TO SEND/RECEIVE: get number from 'indirect_count'
; ADDRESS FOR DATA DERIVED FROM: get address from 'indirect_address'
; OTHER FUNCTIONS:      CALL subroutine listed in command file
; COMMENTS: Option 6 is identical to Option 3, except that a
;           subroutine (whose address is specified in the command
;           file) is called after the data transfer is complete.
;
;option 7 FUNCTION:      send/receive one byte to/from slave from/to 'single_data'
; # BYTES IN THIS COMMAND FILE BLOCK: 2

```

Multimaster I²C routines for byte-oriented I²C interfaces AN435

```

;      NUMBER OF BYTES TO SEND/RECEIVE: 1
;      ADDRESS FOR DATA DERIVED FROM: 'single_data' is addressed
;      OTHER FUNCTIONS: none
;      COMMENTS: Option 7 allows the user to send or receive exactly one
;                  data byte to/from the slave using the 'single_data'
;                  register as the target. The calling routine will have to
;                  write the desired data into the 'single_data' register if
;                  a write operation to the slave is desired. This option
;                  requires very few command file bytes since count and
;                  address information are not needed.
;
;option 8 FUNCTION: send/receive one byte to/from slave from/to 'single_data'
;      # BYTES IN THIS COMMAND FILE BLOCK: 4
;      NUMBER OF BYTES TO SEND/RECEIVE: 1
;      ADDRESS FOR DATA DERIVED FROM: 'single_data' is addressed
;      OTHER FUNCTIONS: new subcall subroutine listed in command file
;      COMMENTS: Option 8 is identical to Option 7, except that a
;                  subroutine (whose address is specified in the command
;                  file) is called after the data transfer is complete.
;
;
;IIC_OS Command File Block Format:
;*****
;NOTE: uppercase 'OR' = logical or function in the following text.
;
;byte # 1 = 7 bit address of slave OR 'iicreadmask' (or 'iicwritemask')
;=====
;
;byte # 2 :
;=====
;
;      option 1 = address space code (which memory space transmit data is read
;                                  from or which memory space receive data is
;                                  written to is specified by an "address
;                                  space code"-- see EQUates in main program.)
;
;      option 2 = 'immediate_' control codes
;                  ('immediate_' indicates that the next byte is the actual data
;                  to be transmitted. This of course is only valid
;                  when writing a byte to a slave. This control
;                  code will save the bytes of information required
;                  to specify the address of the data to be
;                  transmitted. It is a very handy and
;                  efficient mechanism for setting-up the read or
;                  write address in an I2C memory device.)
;
;      option 3 = address space code OR 'indirect_' control code
;                  ('indirect_' indicates that the address for the bytes to be
;                  transmitted or received is not in the command file
;                  but is contained in the IIC_OS register
;                  'indirect_adrs'; also, the number of bytes to be
;                  transmitted or received is contained in the IIC_OS
;                  register 'indirect_count'. The calling routine must
;                  preload these registers, or they must be correctly
;                  loaded from a previous "call_" initiated in the
;                  command file stream.)
;
;      option 4 = address space code OR 'call_' control code
;                  ('call_' indicates that the address of the subroutine to be
;                  called after the present IIC transmission or reception
;                  is complete is contained in the bytes following.)
;

```

Multimaster I²C routines for byte-oriented I²C interfaces AN435

```

; option 5 = 'immediate_' control code OR 'call_' control code
;
; option 6 = address space code OR 'indirect_' OR 'call_'
;
; option 7 = 'singleD_'
; ('singleD_' indicates that one byte is to be read/written, and
; that the target byte to be read/written is the IIC_OS_
; byte called 'single_data'.))
;
; option 8 = 'singleD_' OR 'call_'
;
;byte # 3 :
;=====
; option 1 = number of bytes to be transmitted or received (1 to 255)
;
; option 2 = the data to be transmitted (the 'immediate_' control code was
; used in byte # 2)
;
; option 3 = 'iicend' control code to end session, or next block's byte #1
;
; option 4 = same as option 1
;
; option 5 = same as option 2
;
; option 6 = low address of subroutine to be called after this transmit or
; receive session
;
; option 7 = same as option 3
;
; option 8 = same as option 6
;
;byte # 4 :
;=====
; option 1 = low address of data to be transmitted or received
;
; option 2 = 'iicend' control code to end session, or next block's byte #1
;
; option 3 = Not Applicable
;
; option 4 = same as option 1
;
; option 5 = low address of subroutine to be called after this transmit or
; receive session
;
; option 6 = high address of subroutine to be called after this transmit or
; receive session
;
; option 7 = Not Applicable
;
; option 8 = same as option 6
;
;byte # 5 :
;=====
; option 1 = high address of data to be transmitted or received
; ('iicend' if target memory space is DATA)
;
; option 2 = Not Applicable
;
; option 3 = Not Applicable
;
; option 4 = same as option 1

```

Multimaster I²C routines for byte-oriented I²C interfaces AN435

```

; option 5 = high address of subroutine to be called after this transmit or
; receive session
; option 6 = 'iicend' control code to end session, or next block's byte #1
; option 7 = Not Applicable
; option 8 = same as option 6
;byte # 6 :
;=====
; option 1 = 'iicend' control code to end session, or next block's byte #1
; (Not Applicable if target memory space is DATA)
; option 2 = Not Applicable
; option 3 = Not Applicable
; option 4 = low address of subroutine to be called after this transmit or
; receive session
; option 5 = 'iicend' control code to end session, or next block's byte #1
; option 6 = Not Applicable
; option 7 = Not Applicable
; option 8 = Not Applicable
;byte # 7 :
;=====
; option 1 = Not Applicable
; option 2 = Not Applicable
; option 3 = Not Applicable
; option 4 = high address of subroutine to be called after this transmit or
; receive session
; option 5 = Not Applicable
; option 6 = Not Applicable
; option 7 = Not Applicable
; option 8 = Not Applicable
;byte # 8 :
;=====
; option 1 = Not Applicable
; option 2 = Not Applicable
; option 3 = Not Applicable
; option 4 = 'iicend' control code to end session, or next block's byte #1
; option 5 = Not Applicable

```

Multimaster I²C routines for byte-oriented I²C interfaces AN435

```

; option 6 = Not Applicable
;
; option 7 = Not Applicable
;
; option 8 = Not Applicable
;
;To efficiently use this system, several of these blocks can be put together.
;In fact, there is no limit on the number of blocks allowed.
;This IIC_OS lends itself very nicely to complex IIC requirements. The
;following examples will illustrate the usefulness of this program.
;
;EXAMPLES
;*****
;The following examples are samples for each option. There are so many
;variations that only one version for options 1 - 4 are presented. The code
;fragment "PROGRAM" is simply the part of the code that sets up and calls
;the "WAIT_IIC" program which waits for the execution of the command file.
;The "Optionx_file" code fragments are the code space command files. All
;of the examples show the command file residing in the code space, but they
;could just as easily have been loaded into the DATA or XDATA spaces.
;
;It should be noted that "IIC_OS" will process a command file until an
;'iicend' character is encountered - after which, a STOP condition will be
;implemented. This means that the master can keep possession of the bus
;for as long as it has to.
;
;It is assumed that all the slave addresses and other EQUates have been
;defined in the program previously.
;
;EXAMPLE Option 1
;+++++
;PROGRAM:
;      MOV      IIC_Command_File_adrs,#LOW(Option1_file)      ;load address of
;      MOV      IIC_Command_File_adrs+1,#HIGH(Option1_file)   ;command file
;      CALL     WAIT_IIC_Code      ;call program to wait for IIC execution
;      JMP      MORE_PROGRAM
;
;      ;
;      ;Notice the block structure of the command file. Each block has
;      ;been spaced to accentuate this structure.
;      ;'Option1_file' tells the IIC_OS (called through "WAIT_IIC_Code") to
;      ;read 5 bytes of data in from 'slave1' and store the bytes in the
;      ;DATA space starting at location 'iic_input_data'; after this input
;      ;is done, 'slave2' has 3 bytes written to it from the DATA space
;      ;starting at address 'iic_input_data'.
;      ;Both blocks below are option 1 types, but the first is a read
;      ;and the second is a write.
;      ;
;      ;Option1_file:
;      DB      slave1_address OR iicreadmask      ;slave1 address + read bit
;      DB      ioD_                                ;indicate DATA space target
;      DB      5                                  ;indicate number of bytes
;      DB      iic_input_data                      ;start address of target bytes
;
;      DB      slave2_address OR iicwritemask      ;slave2 address + write bit
;      DB      ioD_                                ;indicate DATA space target
;      DB      3                                  ;indicate number of bytes
;      DB      iic_input_data                      ;start address of target bytes
;
;      DB      iicend                              ;end of iic session
;
;MORE_PROGRAM:

```



```

; continue with program
;
;EXAMPLE Option 2
;+++++
;PROGRAM:
; MOV IIC_Command_File_adrs,#LOW(Option2_file) ;load address of
; MOV IIC_Command_File_adrs+1,#HIGH(Option2_file) ;command file
; CALL WAIT_IIC_Code ;call program to wait for IIC execution
; JMP MORE_PROGRAM
;
; ;Notice the block structure of the command file. Each block has
; ;been spaced to accentuate this structure.
; ;'Option2_file' tells the IIC_OS (called through "WAIT_IIC_Code") to
; ;write one byte of data to the addressed slave - the data is present
; ;in the command file. This action takes 3 bytes in the command file.
; ;In this example, the address for a memory location in an IIC memory
; ;peripheral will be set and the following block does an option 1
; ;type of input.
; ;
;Option2_file:
; DB slave_address OR iicwritemask ;slave address + write bit
; DB immediate_ ;send out next byte only
; DB 1 ;data byte to be sent
; ;
; ;(end of option 2 block)
; DB slave_address OR iicreadmask ;slave address + read bit
; DB ioD_ ;read in 1 byte
; DB 6 ;memory space where bytes go to
; DB iic_input_data + 1 ;number of bytes to be read
; ;
; ;address of input target
; ;
; ;
; DB iicend ;
;
;MORE_PROGRAM:
; continue with program
;
;EXAMPLE Option 3
;+++++
;PROGRAM:
; MOV indirect_adrs,#LOW(input_data1)
; MOV indirect_adrs+1,#HIGH(input_data1)
; MOV indirect_count,#6
; MOV A,decision
; JZ PROGRAM_10
; MOV indirect_adrs,#LOW(input_data2)
; MOV indirect_adrs+1,#HIGH(input_data2)
; MOV indirect_count,#3
;
;PROGRAM_10:
; MOV IIC_Command_File_adrs,#LOW(Option3_file) ;load address of
; MOV IIC_Command_File_adrs+1,#HIGH(Option3_file) ;command file
; CALL WAIT_IIC_Code ;call program to wait for IIC execution
; JMP MORE_PROGRAM
;
; ;
; ;Notice the block structure of the command file. Each block has
; ;been spaced to accentuate this structure.
; ;'Option3_file' tells the IIC_OS (called through "WAIT_IIC_Code") to
; ;read 'indirect_count' number of bytes into external ram space
; ;starting at address 'indirect_adrs'. In the body of "PROGRAM"
; ;the 'indirect' registers are loaded based on a decision. In this
; ;case, if the data byte 'decision' is zero, 6 bytes of data are
; ;read from the slave and placed in external ram starting at the

```

Multimaster I²C routines for byte-oriented I²C interfaces AN435

```

; ;address 'input_data1'; if 'decision' is not zero, three bytes of
; ;data are read from the slave and placed in external ram starting
; ;at address 'input_data2'.
;
;Option3_file:
;   DB    slave_address OR iicreadmask    ;slave address + read bit
;   DB    ioX_OR indirect_name            ;read from external ram area and
;   DB    iicend                          ;use indirect registers
;
;MORE_PROGRAM:
;   continue with program
;
;EXAMPLE Option 4
;+++++
;PROGRAM:
;   MOV    IIC_Command_File_adrs,#LOW(Option4_file) ;load address of
;   MOV    IIC_Command_File_adrs+1,#HIGH(Option4_file) ;command file
;   CALL   WAIT_IIC_Code                      ;call program to wait for IIC execution
;   JMP    MORE_PROGRAM
;
; ;
; ;Notice the block structure of the command file. Each block has
; ;been spaced to accentuate this structure.
; ;'Option4_file' tells the IIC_OS (called through "WAIT_IIC_Code") to
; ;read in 4 bytes from slave1 into data area 'iic_input_data' then
; ;make a call to the subroutine "op4_sub", then output 1 byte of
; ;data from 'single_data' to slave2. The output data was
; ;manipulated by the called subroutine. All this occurred without
; ;a stop condition being generated, so the bus was retained for
; ;the entire period of time.
;
;Option4_file:
;   DB    slave1_address OR iicreadmask    ;slave address + read bit
;   DB    ioD_OR call_                     ;indicate DATA space then call
;   DB    4                                ;indicate number of bytes
;   DB    iic_input_data                   ;start address of target bytes
;   DB    LOW(op4_sub)                     ;address of routine to be
;   DB    HIGH(op4_sub)                    ;executed after read done
;   DB    slave2_address OR iicwritemask    ;slave2 address + write bit
;   DB    singleD_                         ;indicate 'single_data' to be
;   DB    iicend                           ;output (see option 7)
;   DB    ;                                ;end of iic session
;   ;
;   ;Subroutine "op4_sub" adds the first 4 bytes of 'iic_input_data'
;   ;and puts answer into 'single_data'.
;
;op4_sub:
;   MOV    R0,#iic_input_data
;   MOV    R1,#4
;   CLR    A
;
;op4_loop:
;   ADD    A,@R0
;   INC    R0
;   DJNZ   R1,op4_loop
;   MOV    single_data,A
;   RET
;
;MORE_PROGRAM:
;   continue with program

```

Multimaster I²C routines for byte-oriented I²C interfaces AN435

```

;EXAMPLE Option 7
;+++++
;PROGRAM:
;
;   MOV     IIC_Command_File_adrs,#LOW(Option7_file) ;load address of
;   MOV     IIC_Command_File_adrs+1,#HIGH(Option7_file) ;command file
;   CALL    WAIT_IIC ;call program to wait for IIC execution
;   JMP     MORE_PROGRAM
;
; ;
; ;Notice the block structure of the command file. Each block has
; ;been spaced to accentuate this structure.
; ;'Option7_file' tells the IIC_OS (called through "WAIT_IIC_Code") to
; ;read one byte of data from slave1 - the data is placed in
; ;'single_data'. This action takes 2 bytes in the command file.
; ;The next block writes this byte to slave2 using the same option.
;
; ;
;Option2_file:
;   DB      slave1_address OR iicreadmask ;slave address + read bit
;   DB      singleD_ ;read into 'single_data'
;
;   DB      slave2_address OR iicwritemask ;slave address + write bit
;   DB      singleD_ ;get data from 'single_data'
;
;   DB      iicend
;
;MORE_PROGRAM:
;   continue with program
;
;EXAMPLE Option 5 & 6 & 8
;+++++
;Not much more understanding gained by an example - see option 4 for doing
;a call in conjunction with any other option.
;
;EXAMPLE using other data spaces
;*****
;This program can be used such that the command file resides in the internal
;DATA space or the external DATA space. Examples demonstrating the utility
;of this feature will be described below.
;
;DATA Command File Example
;+++++
;
; ;
; ;
; ;change starting address to read from
; ;change number of bytes to read
; ;this call to "SUB" will read 3 bytes from the
; ;IIC data file
;iic_data: DS 8
;datafile: DS 5 ;define a space for the command file
;
; ;
; ;change starting address to read from
; ;change number of bytes to read
; ;this call to "SUB" will read the number of bytes
; ;load the command file into the DATA space.
; ;
;PROGRAM:
;   MOV     datafile,#(slave1_address OR iicreadmask)
;   MOV     datafile+1,#ioD_
;   MOV     datafile+2,#8
;   MOV     datafile+3,#iic_data
;   MOV     datafile+4,#iicend
;
; ;
;P10:
;   MOV     IIC_Command_File_adrs,#datafile ;load address of command file

```

Multimaster I²C routines for byte-oriented I²C interfaces AN435

```

; (notice only 1 byte required)
CALL    WAIT_IIC_Data    ;call program to wait for IIC execution
;
;The IIC commands will be executed from the DATA space starting at 'datafile'
;This may not seem too useful, but it can be a very handy mechanism for
;communicating to a slave which is used often. For example, assume that a
;system uses a PCF8570 IIC RAM for parameter storage. The data in the RAM
;is required often, but the data required may be at any address in the RAM,
;and may be variable in length. Using the DATA space as the command file,
;the programmer could construct a simple and compact mechanism for handling
;this system requirement:
;
;
;
;DSEG
;iic_data:    DS      8                ;IIC data file
;datafile:    DS      8                ;define a space for the command file
;
;
;
;CSEG
;
;
;load the command file into the DATA space. The first block sets the
;subaddress for subsequent access to the RAM; the second block reads
;the RAM.
;
;
;PROGRAM:
;    MOV        datafile,#PCF8570_address OR iicwritemask)
;    MOV        datafile+1,#immediate_
;    MOV        datafile+2,#0
;
;    MOV        datafile+3,#(PCF8570_address OR iicreadmask)
;    MOV        datafile+4,#ioD_
;    MOV        datafile+5,#8
;    MOV        datafile+6,#iic_data
;    MOV        datafile+7,#iicend
;
;P10:
;    CALL        SUB        ;1st call will read 8 bytes from PCF8570 starting at
;                           ;location 0 - bytes will be read into 'iic_data'
;
;
;    MOV        datafile+2,#7        ;change starting address to read from
;    MOV        datafile+5,#3        ;change number of bytes to read
;    CALL        SUB        ;this call to "SUB" will read 3 bytes from the
;                           ;PCF8570 starting at location 7 - bytes will be
;                           ;read into 'iic_data'
;
;    MOV        datafile+2,#0        ;change starting address to read from
;    MOV        datafile+5,#7        ;change number of bytes to read
;    CALL        SUB        ;this call to "SUB" will read the number of bytes
;                           ;specified in R7 from the PCF8570 starting at the
;                           ;location specified by the DATA byte pointed to
;                           ;by R0 - bytes will be read into 'iic_data'
;
;    MOV        datafile+2,#33       ;change starting address to read from
;    MOV        datafile+5,#1        ;change number of bytes to read
;    MOV        datafile+6,#B        ;change target internal address to register B
;    CALL        SUB        ;this call to "SUB" will read 1 byte from the
;                           ;PCF8570 starting at location 33 - the byte will
;                           ;be placed in register B

```

Multimaster I²C routines for byte-oriented I²C interfaces AN435

```

;SUB:
;      MOV      IIC_Command_File_adrs,#datafile ;load address of command file
;      CALL     WAIT_IIC_Data ;call program to wait for IIC execution
;      RET
;One can construct a command file such that the called subroutine actually
;modifies the command file itself! Also, the called subroutine could modify
;the 'IIC_Command_File_adrs' register so that upon returning from the
;subroutine, a different IIC command file is executed other than the one
;immediately after the block that called the subroutine.
;XDATA Command File Example
;+++++
;
;      .
;      .
;      .
;DSEG
;iic_data:      DS      8 ;IIC data file
;XSEG
;datafile:      DS      5 ;define a space for the command file
;
;      .
;      .
;CSEG
;
;      ;
;      ;load the command file into the XDATA space.
;
;PROGRAM:
;      MOV      DPTR,#datafile
;      MOV      A,#(slavel_address OR iicreadmask)
;      MOVX     @DPTR,A
;      INC      DPTR
;      MOV      A,#ioD
;      MOVX     @DPTR,A
;      INC      DPTR
;      MOV      A,#8
;      MOVX     @DPTR,A
;      INC      DPTR
;      MOV      A,#iic_data
;      MOVX     @DPTR,A
;      INC      DPTR
;      MOV      A,#iicend
;      MOVX     @DPTR,A
;
;P10:
;      MOV      IIC_Command_File_adrs,#LOW(datafile) ;load address of
;      MOV      IIC_Command_File_adrs+1,#HIGH(datafile) ;command file
;      CALL     WAIT_IIC_Xdata ;call program to wait for IIC execution
;
;      .
;      .
;This program will run the command file from external data memory. All the
;same options exist with XDATA command files as do with DATA command files.
;-----
;SYSTEM REQUIREMENTS:
;*****
;Data Bytes Used: 15
;Bit Addressable Bytes Used: 1
;Bits used: 2
;Stack Penetration: Approximately 17 bytes worst case
;Code Length: Approximately 600 bytes of code.
;NOTE: most of the code length of this program is made up of

```


Multimaster I²C routines for byte-oriented I²C interfaces AN435

```

; code used to take care of every generic addressing
; possibility. If the user simply wants to use one known
; address space to hold the command file, and one target
; I/O address in a specific data space, the code and
; data requirements would be reduced dramatically. This
; code is a good starting point for custom development
; since it is completely generic.
;
; "IIC_OS" register definitions:
;*****
;The following data bytes are required to run the full implementation of the
;IIC driver system. In a microcontroller as large as the 80C552 or
;80C652, the requirement of these data bytes will not impose a great toll
;on the user's system. Note that registers, bytes, equates and bit names are
;surrounded by ' ' in the description - routines, subroutines and procedure
;names are surrounded by " ".
;
;-----
; 'IIC_status'
;
;LOADED BY: "WAIT_IIC" and "IIC_VECTOR"
;DESCRIPTION - holds the status of the requested IIC operations. This data
;byte is loaded with 'status_DO_IIC' by "WAIT_IIC", which then
;monitors this byte, and determines if the IIC command file has been
;completed. Completion of the IIC command file is known if the 'IIC_status'
;is equal to one of the following:
;
; 'status_OK' - operation complete, no problems
; 'status_arb_lost' - arbitration lost to another master
; 'status_attempt_data' - tried to send data 'max_data_attempts' times
; 'status_attempt_adrs' - tried to find slave 'max_adrs_attempt' times
; 'status_timeout' - waited 'max_wait' time for activity
; 'status_buss_err' - a buss error (illegal start/stop)
; 'status_slave' - addressed as slave (own address)
; 'status_arb_lost_slave' - arbitration lost to another master, this one
; addressed as a slave
; 'status_general_slave' - addressed as slave (general call)
; 'status_arb_lost_general' - arbitration lost to a general call
;
;The values 'max_data_attempts', 'max_adrs_attempts' and 'max_wait' are
;equated in the main body below - these numbers define how many attempts
;should be made to send/receive data, locate a slave or wait for a response.
;
;-----
; 'IO_buffer_adrs'
;
;LOADED BY: "IIC_OS"
;DESCRIPTION: is loaded by the "IIC_OS" operation and holds the address
;of the data to be transmitted to a slave, or received from a slave. The bit-
;addressable register 'Command?adrs?space' determines which memory space the
;'IO_buffer_adrs' targets. The initial value for this register can come from the
;the command file itself, or may be loaded from the 'indirect_adrs' register,
;depending on the actions directed from the 'Command?adrs?space' register.
;
;-----
; 'IIC_Command_File_adrs'
;
;LOADED BY: calling routine, manipulated by "IIC_OS"
;DESCRIPTION: the calling routine loads the address of the command file to be
;executed by the "IIC_OS" into this two byte register. The "IIC_OS" will
;modify this address register as the IIC OS operations proceed. If the command

```

Multimaster I²C routines for byte-oriented I²C interfaces AN435

```

;file resides in the DATA space, then only the LSByte of the address is used.
;
;-----
;indirect_adrs'
;indirect_count'
;
;LOADED BY: calling routine or a command file called subroutine
;DESCRIPTION: the calling routine may use the 'indirect_' option as
;loaded in the 'Command?adrs?space' byte. This option directs the "IIC_OS"
;to get the 'IO_buffer_adrs' and the 'Multiple_count' from the calling routine
;(or called subroutine) loaded 'indirect_adrs' and 'indirect_count' registers.
;
;-----
;'single_data'
;
;LOADED BY: calling routine or "IIC_VECTOR"
;DESCRIPTION: this byte allows for a quick mechanism to input one single byte
;of data from the IIC bus into the DATA space, or send 1 byte of data from the
;DATA space to the IIC bus. If the IIC command file has the control code
;'singleD_', then the bit 'singleDATA' will be set in 'Command?adrs?space'.
;If 'singleDATA' is set, then the system will either input one byte of data
;only into 'single_data' or output the contents of 'single_data'.
;
;-----
;'Multiple_count'
;
;LOADED BY: "IIC_OS"
;DESCRIPTION: this register is a counter for the number of bytes to be
;received or transmitted. Received or transmitted bytes are sent to or read
;from the address space indicated in 'Command?adrs?space' and addressed by
;the 'IO_buffer_adrs'. The initial value for this register can come from
;the command file itself, or may be loaded from the 'indirect_count' register,
;depending on the actions directed from the 'Command?adrs?space' register.
;
;-----
;'Attempt_count'
;
;LOADED BY: "IIC_OS"
;DESCRIPTION: counts the number of failed attempts at sending/receiving data
;or addressing a slave. If the number of tries in either case exceeds
;'max_adrs_attempts' or 'max_data_attempts', the error status is reflected in
;'IIC_status' and the "IIC_OS" quits.
;
;-----
;'last_data'
;
;LOADED BY: "IIC_OS"
;DESCRIPTION: holds the value of the last data byte received or transmitted.
;Used in "IIC_OS" as a look-back register so failed transmissions can be
;repeated.
;
;-----
;'iic_timer'
;
;LOADED BY: "IIC_OS"
;DESCRIPTION: used as a watchdog timer for the IIC operation. Implemented as
;a up-counter in "WAIT_IIC", but ideally, this function should be in the
;hands of a real system timer.
;
;-----
;'Slave_in & Slave_out'

```

Multimaster I²C routines for byte-oriented I²C interfaces AN435

```

;
;LOADED BY: mainline routine
;DESCRIPTION: Buffers for Slave receiver and Transmitter modes. Main routine
;loads 'Slave_out' with the SLVbytes_out' number of bytes to be transmitted
;once addressed by another master. 'Slave_in' is filled by the 'SLVbytes_in'
;number of bytes from another master. If more or less bytes are to be received
;or sent when addressed as a slave, then the size of the buffers and the
;'SLVbytes'... EQUates must change.
;
;IIC_OS does not need these 16 DATA bytes if the system is single-master.
;IF your system is single-master, then use the 'Slave_in' and 'Slave_out'
;buffers for your own general purpose buffers or registers.
;
;-----
;'temp'
;
;This byte is used in the "WAIT_IIC" routine to hold the appropriate data
;space temporarily until the IIC bus is free or a slave receiver or slave
;transmitter mode is done.
;
;Slave address for this microcontroller - other bus masters can address this
;micro as a slave; this driver simply sends 'SLVbytes_out' number of bytes or
;receives 'SLVbytes_in' number of bytes in the case of being addressed as a
;slave. The LSBit of the address is set indicating that general calls will
;be responded to.
;
Own_adrs      EQU      02EH      ;address of micro when addressed as a slave
general_enable EQU      1        ;general call recognized since LSBit is set
SLVbytes_in   EQU      8        ;# bytes to receive when addressed as a slave
SLVbytes_out   EQU      8        ;# bytes to transmit when " " " "
DSEG         AT          48      ;change location to suit your system

IIC_status:   DS          1
IO_buffer_adrs: DS        2
IIC_Command_File_adrs: DS      2
indirect_adrs: DS          2
indirect_count: DS         1
iic_timer:    DS          2
Attempt_count: DS          1
Multiple_count: DS         1
last_data:    DS          1
single_data:  DS          1
temp:         DS          1

Slave_in:     DS          SLVbytes_in
Slave_out:    DS          SLVbytes_out

DATA_start    EQU      Slave_out+SLVbytes_out
;
;'Command?adrs?space' is loaded by the calling routine, and manipulated by the
;"IIC_OS" routine. It indicates which memory space the command file is to
;be read from. It also ultimately gets the address space for the input and
;output data, as well as the indication for the special functions 'indirect',
;'immediate' and 'call'. Generally speaking, the calling routine loads
;'Command?adrs?space' with the code for which memory space holds the command
;file to be executed - then, the command file information amends this byte
;as each block of the command file is executed.
;
DSEG         AT          32      ;change location to suit your system - bit addressable!

```

Multimaster I²C routines for byte-oriented I²C interfaces AN435

```

Command?adrs?space:    DS      1
;address to match 'Command?adrs?space' byte

BSEG AT 0
;address to indicate whether data is a command or a point

command_Data:          DBIT     1
;max_data_attempts' failed to get/send data
command_Code:          DBIT     1
;this system should reply to address a slave
IO_Data:               DBIT     1
;before it gives up and says an error has occurred
IO_Code:               DBIT     1
;this system should reply to address a slave
immediate_data:        DBIT     1
;before it gives up and says an error has occurred
call_function:         DBIT     1
;used to give roughly the timeout time required
indirect_xxx:          DBIT     1
;IIC inactivity timeout. The count will depend
singleDATA:           DBIT     1
;on the clock speed as well as the average loop time in "WAIT_IIC".
;The loop time will be affected by the IIC interrupt processing as
;well as any other interrupts.

BSEG
;
;this next bit is set whenever the microcontroller is addressed as a slave
;receiver or a slave transmitter. "WAIT_IIC" needs this bit to hold off
;pending calls from the main routines to the IIC bus.
;
;_am_a_slave:          DBIT     1
;maximum tries to get/send data
;
;if any iic session failure occurs, set the failure bit
;
IIC_failure:           DBIT     1
;count value for "IIC_FAILURE"

BITS_start EQU IIC_failure+1
;
;mask bytes for the various 'Command?adrs?space' bits. These codes are used
;in the command file. See examples above.
;
commandD_ EQU 00000001B ;commands come from DATA space
commandC_ EQU 00000010B ;commands come from CODE space
commandX_ EQU 00000000B ;commands come from XDATA space
ioD_ EQU 00000100B ;input/output data from DATA space
ioC_ EQU 00001000B ;output data from CODE space
ioX_ EQU 00000000B ;input/output data from XDATA space
immediate_ EQU 00010000B ;next byte is data to be sent
call_ EQU 00100000B ;call subroutine after in/out
indirect_ EQU 01000000B ;get info from 'indirect_registers'
singled_ EQU 10000000B ;get/put 1 byte to/from 'single_data'
commands_ EQU 00000011B ;mask to isolate command adrs bits
iospaces_ EQU 00001100B ;mask to isolate I/O space bits
specials_ EQU 11110000B ;mask to isolate control bits
;
;The following status bytes are used to indicate the present state as well
;as the ultimate state of the IIC operations. These values are loaded into
;'IIC_status' by the various states as well as "WAIT_IIC".
;
status_OK EQU 0 ;end of session and/or bus available
status_arb_lost EQU 1 ;arbitration lost
status_attempt_data EQU 2 ;'max_data_attempts' failed to get/send data
status_attempt_adrs EQU 3 ;'max_adrs_attempts' failed to find slave
status_timeout EQU 4 ;'WAIT_IIC' detected timeout problem
status_buss_err EQU 5 ;a bus error or illegal start/stop
status_slave EQU 6 ;addressed as slave (own address)
status_general_slave EQU 7 ;addressed as slave (general call)
status_arb_lost_slave EQU 8 ;arbitration lost, addressed as slave
status_arb_lost_general EQU 9 ;arbitration lost, general call
status_DO_IIC EQU 0FFH ;all running fine (bus busy) indication

```

Multimaster I²C routines for byte-oriented I²C interfaces AN435

```
;IIC control characters.
;These characters are used in the IIC command file and various routines.
; 'iicend' must be the last character in any command file sequence.
; 'iicwritemask' and 'iicreadmask' are used in conjunction with the slave
; address to indicate whether data is a comin' or a goin'.
; 'max_data_attempts' should be equated to a value indicating how many times
; this system should re-try to get data from/to a slave
; before it gives up and says an error has occurred.
; 'max_adrs_attempts' should be equated to a value indicating how many times
; this system should re-try to address a slave
; before it gives up and says an error has occurred.
; 'max_wait' is used in a crude loop counting timer in "WAIT_IIC". This number
; should be equated to give roughly the timeout time required by
; your system (i.e. IIC inactivity timeout). The count will depend
; on the clock speed as well as the average loop time in "WAIT_IIC".
; The loop time will be affected by the IIC interrupt processing as
; well as any other interrupt service routines in your system.

;
; iicend EQU OFFH ;end of IIC command file
; iicwritemask EQU 00H
; iicreadmask EQU 01H
; max_data_attempts EQU 3 ;maximum tries to get/send data
; max_adrs_attempts EQU 3 ;maximum tries to address slave
; max_wait EQU 1 ;reload value for 'iic_timer'
; ;(counts up)

;
; 'S1STA' reload values. Virtually the same as old '552 app. note.
;
; ENS1_NOTSTA_STO_NOTSI_NOTAA_CRO EQU 0D1H
; ENS1_NOTSTA_STO_NOTSI_AA_CRO EQU 0D5H
; ENS1_NOTSTA_NOTSTO_NOTSI_AA_CRO EQU 0C5H
; ENS1_NOTSTA_NOTSTO_NOTSI_NOTAA_CRO EQU 0C1H
; ENS1_STA_NOTSTO_NOTSI_AA_CRO EQU 0E5H
;
;
; IIC hardware interrupt vector definition.
;
CSEG AT 002BH
JMP IIC_VECTOR ;IIC_OS interrupt service routine
;
;
; CSEG
;
; =====
; "WAIT_IIC" ;The calling routine has loaded the 'IIC_Command_File_adrs'
; register with the address of the command file to be executed.
; This routine simply waits for the IIC process to be completed. Completion
; of the IIC session is indicated by 'IIC_status'='status_OK', or one of the
; many error codes. "IIC_VECTOR" will take care of updating the status byte.
;
;
; The calling routine must call the appropriate section of "WAIT_IIC" depending
; on which memory space the command file resides. For example, if the command
; file resides in CODE memory space, the "WAIT_IIC_Code" must be called.
;
;
; It is possible that another master has addressed this master as a slave. If
; "WAIT_IIC" is called under these circumstances, it will exit with
; 'IIC_failure' set and also check for a timeout for the addressed slave
; session. If a timeout is detected, all IIC_OS registers and bits are set to
; their default value. In either case the 'IIC_failure' bit is set.
;
; In the addressed slave mode, the calling master determines bus timing and
; clock values etc. If something happens to that master, and it cannot
```


Multimaster I²C routines for byte-oriented I²C interfaces AN435

```

;complete it's session, then this slave is left hanging! For this reason,
;we have a built-in timeout check feature for addressed slave mode (if and
;when this slave calls "WAIT_IIC" to do it's own work).
;
;INPUT:
;=====
;'IIC_status' is checked to ensure an addressed slave state is not in progress
;OUTPUT:
;=====
;'IIC_status' is updated to reflect completion of IIC session or error.
;'IIC_failure' is updated (0 = all OK, 1 = some kind of error).
;=====
;
WAIT_IIC_Data:
    MOV     temp,#commandD_
    SJMP    WAIT_IIC
WAIT_IIC_Xdata:
    MOV     temp,#commandX_
    SJMP    WAIT_IIC
WAIT_IIC_Code:
    MOV     temp,#commandC_
WAIT_IIC:
    JNB     i_am_a_slave,WIIC_10
;
;if a slave receive or transmit session is in effect (as initiated by
;another master), this processor will only know that through the
;"IIC_VECTOR" routine - there is no timeout check etc. Because of
;this situation, "WAIT_IIC" will check for a slave session in
;progress and exit as a failure if all is OK with that session (so
;that the calling routine will keep trying to get hold of the bus).
;If the slave session has timed out or there is an error, clear
;everything and exit as an error as well.
;
WIIC_05:
    CALL    IIC_time                ;addressed slave timeout check
    JNZ     WIIC_35                 ;if not, exit as a failure
    WII_06: CLR i_am_a_slave        ;if timeout, reset system and exit
    SJMP    WIIC_ERR               ;as a failure
;
;ready to try - this micro is not presently addressed as a slave, and
;all else seems to be OK.
WIIC_10:
    MOV     Command?adrs?space,temp
    MOV     IIC_status,#status_DO_IIC ;indicate IIC busy
    SETB    STA                    ;do an IIC interrupt (start start condition)
WIIC_15:
    MOV     iic_timer,#LOW(max_wait) ;start timeout timer
    MOV     iic_timer + 1,#HIGH(max_wait)
WIIC_20:
    CALL    IIC_time
    JZ      WIIC_ERR
;
;as long as 'iic_timer' is OK, loop here and check the 'IIC_status'.
;'IIC_status' will remain as 'status_DO_IIC' as long as the IIC
;session is still on. This byte will be loaded with 'status_OK' if
;the session ends normally, or it will be loaded with some other
;status byte if an error or arbitration process occurs.
;
;If this micro has been addressed as a slave, or has lost arbitration
;and become a slave, then 'IIC_status' indicates the situation, and

```

Multimaster I²C routines for byte-oriented I²C interfaces AN435

```

;this subroutine is terminated. The routine that calls this one must
;check the 'IIC_status' to determine if another master has won the bus
;so that it can wait for 'IIC_status' to become 'status_OK', at which
;point, it could try again.
;
WIIC_25:
    MOV     A,IIC_status
    CJNE    A,#status_DO_IIC,WIIC_30
    SJMP     WIIC_20

WIIC_30:
    CJNE    A,#status_OK,WIIC_35

WIIC_X:
    CLR     IIC_failure
    CLR     i_am_a_slave
    RET

;
;if 'iic_timer' overflows, have an IIC bus timeout error.
;
WIIC_ERR:
    CALL    MORE_00_SUB
    ANL     IIC_status, #11110000B
    ORL     IIC_status, #status_timeout
    CALL    WIIC_ERRX

;
;do error recovery here - i.e. lost arbitration, bus error.
;Alternately, can return the error code to the calling routine so
;that the main routines decide what to do for various errors. For
;development and debug purposes, this example routine ignores the
;errors.
;
WIIC_35:
    SETB    IIC_failure
    RET

WIIC_ERRX:
    RETI

;=====
; "IIC_time"
; Subroutine to increment IIC timeout timer. ACC returns 0 if timeout
; occurs.
;
IIC_time:
    INC     iic_timer
    MOV     A,iic_timer
    JNZ     time_X
    INC     iic_timer + 1
    MOV     A,iic_timer + 1
    time_X:
    RET

;
;=====
; Subroutine "FETCH_DATA"
; DESCRIPTION:
;=====
; This subroutine is used by all the IIC_OS states to get the next data byte
; from the address 'IO_address' in the memory space indicated in the
; 'Command?adrs?space'. This routine also saves the fetched data in the byte
; 'last_data' so error recovery can be easily done. Before this routine exits,
; the pointer 'IO_buffer_adrs' is incremented.
; Fetched data returned in ACCumulator.
; INPUT:

```

Multimaster I²C routines for byte-oriented I²C interfaces AN435

```

=====
; 'IO_address' has address where data is to be fetched from. If the target
; space is DATA, then the LSByte of 'IO_address' is the full
; address and the MSByte is ignored.
; 'Command?adrs?space' holds the information for which address space is to be
; targeted for fetching the data.
; OUTPUT:
; =====
; Accumulator is loaded with the fetched byte.
; 'last_data' gets the fetched byte as well.
; 'IO_address' is incremented.
;
FETCH_DATA:
        JB      IO_Data,FD_Data ;is data in DATA space?
        MOV     DPL,IO_buffer_adrs ;no, then must be XDATA or CODE space
        MOV     DPH,IO_buffer_adrs+1
        JB      IO_Code,FD_Code ;is it in CODE space?
;enter here if the target space is DATA

FD_Xdata:
        MOVX    A,@DPTR ;no, it must be in XDATA space
;use the address in R0
        MOV     R0,IO_buffer_adrs
        MOV     R0,A
        MOV     last_data,A ;store data
        INC     DPTR ;bump pointer
        MOV     IO_buffer_adrs,DPL ;restore pointer
        MOV     IO_buffer_adrs + 1,DPH
        RET
;
;enter here if data is in CODE space
;
FD_Code:
        CLR     A
        MOVC    A,@A+DPTR
        SJMP     FD_exit
;
;enter here if data is in DATA space
;
FD_Data:
        MOV     R0,IO_buffer_adrs
        MOV     A,@R0
        MOV     last_data,A
        INC     R0
        MOV     IO_buffer_adrs,R0
        RET
;
;=====
;Subroutine "STORE_DATA"
;DESCRIPTION:
;=====
;This subroutine stores incoming data in the address 'IO_address' in the
;data space indicated in 'Command?adrs?space'. Only XDATA and DATA spaces
;are valid since we cannot write into the CODE space.
; INPUT:
; =====
; Accumulator has data to be stored. This data is not corrupted.
; 'IO_address' holds address where data is to be stored
; 'Command?adrs?space' (bit addressable) describes which data space is to
; be targeted.
; OUTPUT:
; =====
; Accumulator contents not corrupted by subroutine

```

Multimaster I²C routines for byte-oriented I²C interfaces AN435

```

;ACCumulator contents are stored in address as described above.
;'last_data' holds a copy of the data.
;'IO_address' is incremented.
;
STORE_DATA:
    JB      IO_Data,SD_Data          ;is target area DATA?

SD_Xdata:
    MOV     DPL,IO_buffer_adrs        ;no, then must be XDATA so load
    MOV     DPH,IO_buffer_adrs+1      ;address into DPTR
    MOVX    @DPTR,A                  ;store the data
    MOV     last_data,A               ;and copy it
    INC     DPTR                      ;bump the address pointer
    MOV     IO_buffer_adrs,DPL        ;and restore it
    MOV     IO_buffer_adrs+1,DPH
    RET

;
;enter here if the target space is DATA

SD_Data:
    MOV     R0,IO_buffer_adrs        ;get the address into R0
    MOV     @R0,A                    ;store the data
    MOV     last_data,A               ;and copy it
    INC     R0                        ;bump the address pointer
    MOV     IO_buffer_adrs,R0
    RET

;
;=====
;Subroutine "FETCH_COMMAND"
;DESCRIPTION:
;=====
;This subroutine fetches a byte from the address 'IIC_Command_File_adrs' in
;the address space indicated in 'Command?adrs?space' (bit addressable). If
;"FETCH_COMMAND_0" is called, then the address pointer 'IIC_Command_File_adrs'
;is not incremented at exit, otherwise the address pointer is incremented.
;INPUT:
;=====
;'IIC_Command_File_adrs' holds the address of the byte to be retrieved.
;'Command?adrs?space' indicates in which address space the command file resides
;OUTPUT:
;=====
;ACCumulator holds the retrieved byte.
;'IIC_Command_File_adrs' is incremented (not if "FETCH_COMMAND_0" is called).
;
FETCH_COMMAND_0:
    CLR     C                        ;carry indicates whether pointer inc or not
    SJMP    FC_10

FETCH_COMMAND:
    SETB    C

FC_10:
    JB      command_Data,FC_Data      ;is command file in DATA space?
    MOV     DPL,IIC_Command_File_adrs ;no, must be XDATA or CODE
    MOV     DPH,IIC_Command_File_adrs+1
    JB      command_Code,FC_Code      ;is command file in CODE space?

FC_Xdata:
    MOVX    A,@DPTR                  ;no, then must be in XDATA

FC_exit:

```

Multimaster I²C routines for byte-oriented I²C interfaces AN435

```

JNC     FCX_10      ;don't increment pointer if no carry
INC     DPTR        ;if carry set, increment pointer
FCX_10:
MOV     IIC_Command_File_adrs,DPL      ;restore pointer
MOV     IIC_Command_File_adrs+1,DPH
RET
;
;enter here to retrieve command byte from CODE space
;
FC_Code:
CLR     A           ;get SFR indicating IIC hardware status for the 'S1STA'
MOVC    A,@A+DPTR    ;limited range of values, namely 00H to 00H in steps of 08H. The following
SJMP    FC_exit      ;manipulation changes the 'S1STA' value to a number from 0 to 32. This
;
;enter here to retrieve command byte from DATA space
;
FC_Data:
MOV     R0,IIC_Command_File_adrs
MOV     A,R0
JNC     FCD_X
INC     R0
FCD_X:
MOV     IIC_Command_File_adrs,R0
RET
;
;=====
;Subroutine "make_space"
;DESCRIPTION:
;=====
;"make_space" is used to update the 'Command?adrs?space' byte which holds
;the information for which memory space is to be targeted for data and
;command bytes. This subroutine is usually called by a state that has just
;read-in the command file byte indicating address space information.
;INPUT:
;=====
;ACCumulator has the data address space indication read-in from command file.
;OUTPUT:
;=====
;'Command?adrs?space' is updated with ACCumulator contents.
;
;=====
make_space:
XCH     A,Command?adrs?space      ;get present address space byte
ANL     A,#commands_              ;clear all bits except command bits
ORL     A,Command?adrs?space      ;mask in new address space info
XCH     A,Command?adrs?space      ;restore
ms_X:
RET
;
$sejct
;
;=====
;IIC interrupt vector
;Every time a significant event occurs on the IIC bus (a start, stop, error,
;etc.), this interrupt routine is entered. This routine reads the IIC
;hardware SFR called 'S1STA' to determine what state the IIC hardware is in.
;Each state has it's own processing routine as shown below.
;The multimaster routines shown are very simple in this module - multimaster
;functions are very dependent on the system being serviced. This
;module simply relinquishes control of the bus if another master wins
;arbitration; it will receive or send bytes if it is addressed as a
;slave.

```


Multimaster I²C routines for byte-oriented I²C interfaces AN435

```

;=====
;
; IIC_VECTOR:
;   PUSH   PSW           ;save all registers used in interrupt vector
;   PUSH   ACC
;   PUSH   DPL
;   PUSH   DPH
;   PUSH   ARO
;
;
; 'S1STA', the SFR indicating IIC hardware status for the '552 takes on a
; limited range of values, namely 00H to 0C8H in steps of 08H. The following
; manipulation changes the 'S1STA' value to a number from 0 to 25. This
; number is then multiplied by 2 so a jump can be done from an 'AJMP' table.
;
;   MOV     A,S1STA       ;get SFR which holds hardware status of bus
;   SWAP    A
;   RLC     A
;   JNC     IICV_10
;   INC     A
;
; IICV_10:
;   RL      A
;   MOV     DPTR,#S1STA_00
;   JMP     @A+DPTR
;
;
; all sections exit here.
; The timeout timer 'iic_timer' is restarted every time around, it is assumed
; that if an interrupt occurs, that more than likely, everything is OK.
;
; IIC_EXIT:
;   MOV     iic_timer,#LOW(max_wait) ;reload timeout timer
;   MOV     iic_timer + 1,#HIGH(max_wait)
;
;   POP     ARO
;   POP     DPH
;   POP     DPL
;   POP     ACC
;   POP     PSW
;   RETI
;
;
; -----
; Jump table for interrupt routine entry above.
;
; S1STA_00:
;   AJMP    MORE_00           ;Bus Error mode
;
;   AJMP    MORE_08           ;Master Receiver/Transmitter Mode
;   AJMP    MORE_10           ;Master Receiver/Transmitter Mode
;   AJMP    MORE_18           ;Master Transmitter Mode
;   AJMP    MORE_20           ;Master Transmitter Mode
;   AJMP    MORE_28           ;Master Transmitter Mode
;   AJMP    MORE_30           ;Master Transmitter Mode
;   AJMP    MORE_38           ;Master Receiver/Transmitter Mode
;
;   AJMP    MORE_40           ;Master Receiver Mode
;   AJMP    MORE_48           ;Master Receiver Mode
;   AJMP    MORE_50           ;Master Receiver Mode
;   AJMP    MORE_58           ;Master Receiver Mode
;
;   AJMP    MORE_60           ;Slave Receiver Mode
;   AJMP    MORE_68           ;Slave Receiver Mode

```

Multimaster I²C routines for byte-oriented I²C interfaces AN435

```

AJMP     MORE_70                ;Slave Receiver Mode
AJMP     MORE_78                ;Slave Receiver Mode
AJMP     MORE_80                ;Slave Receiver Mode
AJMP     MORE_88                ;Slave Receiver Mode
AJMP     MORE_90                ;Slave Receiver Mode
AJMP     MORE_98                ;Slave Receiver Mode
AJMP     MORE_A0                ;Slave Receiver Mode

AJMP     MORE_A8                ;Slave Transmitter Mode
AJMP     MORE_B0                ;Slave Transmitter Mode
AJMP     MORE_B8                ;Slave Transmitter Mode
AJMP     MORE_C0                ;Slave Transmitter Mode
AJMP     MORE_C8                ;Slave Transmitter Mode

;-----
;State 00 = Bus error due to an illegal START or STOP condition. This state
;can also occur if the SIO1 enters an undefined state.
;-----
;
MORE_00:
MOV      IIC_status,#status_buss_err ;indicate bus error
ANL      P1,#00111111B                ;unstick bus
ORL      P1,#11000000B

M00_10:
CALL     MORE_00_SUB ;clear all "IIC_OS" status counters etc.
JMP      IIC_EXIT

;
;This portion of State 00 was made into a subroutine so that the "WAIT_IIC"
;routine could call it when a timeout error occurs.
;This routine sets all counters and other "IIC_OS" registers to 0.
;
;-----
MORE_00_SUB:
CLR      i_am_a_slave
CLR      A
MOV      Attempt_count,A
MOV      Multiple_count,A
MOV      last_data,A
MOV      S1CON,#ENS1_NOTSTA_STO_NOTSI_AA_CR0 ;STOP
RET

;-----
;State 08 indicates that a start condition has been transmitted.
;MASTER RECEIVER/TRANSMITTER MODE.
;In this case, the "IIC_OS" possibilities are one - the next byte in the
;'IIC_Command_File' must be the slave address (and read/write bit).
;-----
;
MORE_08:
CALL     FETCH_COMMAND ;get next byte in command file
MOV      S1DAT,A ;transmit it

M08_10:
MOV      S1CON,#ENS1_NOTSTA_NOTSTO_NOTSI_NOTAA_CR0
JMP      IIC_EXIT

;-----
;State 10H = a repeated start condition has been transmitted.
;MASTER RECEIVER/TRANSMITTER MODE.
;This state is handled just like State 08H. The "IIC_OS" definition ensures
;that the next byte in the 'IIC_Command_File' will be a slave address (and
;read/write bit).
;-----

```

Multimaster I²C routines for byte-oriented I²C interfaces AN435

```

;
; MORE_10:
;       JMP     MORE_08
;
;-----
;State 18H = (Slave address + Write bit) has been transmitted, and an ACK has
;       been returned.
;MASTER TRANSMITTER MODE.
;Once a slave address has been transmitted, several possibilities exist,
;namely: set-up to send/receive n bytes with count and address info coming
;       from the command file stream
;
;       OR
;
;       set-up to send/receive n bytes with count and address info coming
;       from the 'indirect_count' and 'indirect_adrs' registers. The
;       mainline routine must set these registers before initiating an IIC
;       session.
;
;       OR
;
;       set-up to send 1 byte ('immediate_') from the command file stream.
;
;       OR
;
;       set-up to send/receive 1 byte from/to 'single_data' register. The
;       mainline routine must load 'single_data' if it is to be transmitted.
;-----
MORE_18:
MOV     Attempt_count,#0;clear failed attempt count
M18_15:
CALL    FETCH_COMMAND      ;get next byte in command file
CALL    make_space         ;update 'Command?adrs?space'
JBC     immediate_data,M18_30 ;is data immediate?
CALL    M18_SUB            ;No, call subroutine to load count
;and address of data
M18_20:
CALL    FETCH_DATA        ;now ready to get data byte
M18_25:
MOV     S1DAT,A           ;send as data
M18_X:
MOV     S1CON,#ENS1_NOTSTA_NOTSTO_NOTSI_AA_CR0
JMP     IIC_EXIT
;
;enter here if immediate data output requested.
;The data to be transmitted is the next byte in the command file.
;
M18_30:
MOV     Multiple_count,#0
CALL    FETCH_COMMAND
SJMP    M18_25
;
; "M18_SUB" subroutine checks for 'indirect_' and 'singleD_' commands.
;State 18H and State 40H use this subroutine.
;If an indirect feature is requested, load address and count
;information from the 'indirect_count' and 'indirect_adrs' registers
;if the 'indirect_' feature is not requested, then the count and
;address information are contained in the next bytes of the command
;file.
;
M18_SUB:
JBC     indirect_xxx,M18S_10 ;if indirect, clear bit and service
JBC     singleDATA,M18S_20   ;if one data byte in/out to 'iic_data'
;
;enter here if the count and address for the data to be read/written
;is in the command file itself (i.e. no special commands).

```

Multimaster I²C routines for byte-oriented I²C interfaces AN435

```

CALL    FETCH_COMMAND      ;get next byte in command file
DEC     A                  ;decrement and
MOV     Multiple_count,A   ;store as byte count
CALL    FETCH_COMMAND      ;get next byte in command file
MOV     IO_buffer_adrs,A   ;store as LSByte of data address
JB      IO_Data,M18S_X     ;if DATA space, only 1 address byte
CALL    FETCH_COMMAND      ;get next byte in command file
MOV     IO_buffer_adrs+1,A ;store as MSByte of data address
M18S_X:
RET
;
;enter here if indirect requested. The number of bytes to be
;written or read is contained in the 'indirect_count' register,
;the address of the bytes to be read or written is contained in
;the 'indirect_adrs' register(s).
;
M18S_10:
DEC     indirect_count
MOV     Multiple_count,indirect_count
MOV     IO_buffer_adrs,indirect_adrs
MOV     IO_buffer_adrs + 1,indirect_adrs + 1
RET
;
;enter here if single byte input/output from DATA space requested
;('singleDATA').
;
M18S_20:
MOV     Multiple_count,#0
MOV     A,#NOT(iospaces_)
ANL     A,Command?adrs?space
ORL     A,#ioD_
MOV     Command?adrs?space,A
MOV     IO_buffer_adrs,#single_data
RET
;
;-----
;State 20H = (Slave address + Write bit) has been transmitted, no ACK from
;slave.
;MASTER TRANSMITTER MODE.
;This state counts the number of failures for a transmitted address - if
;'max_adrs_attempts' failures occur in-a-row, then abort session.
;-----
;
;The called subroutine could be used to modify the contents of the
;'IIC_Command_File_adrs' registers. In doing so, IF-TURN-HIGH
;control flow could bump attempt count on some IIC read information.
;The subroutine may decide to run one of several other IIC blocks
;for and the session. If the sub fails, the sub will return to the
;used to manipulate
M18S_20:
INC     Attempt_count ;bump attempt count
MOV     A,Attempt_count
CJNE    A,#max_adrs_attempts,M20_10 ;if too many failures,
MOV     IIC_status,#status_attempt_adrs ;indicate attempt error
JMP     M00_10
;
;if less than 'max_attempts' failures, then set command file pointer
;back one, and try sending address again.
;
M20_10:
MOV     R0,#IIC_Command_File_adrs
MOV     A,@R0
JNZ     M20_20
INC     R0
DEC     @R0
DEC     R0
M20_20:

```

Multimaster I²C routines for byte-oriented I²C interfaces AN435

```

DEC      @R0
JMP      M08_10
;-----
;State 28H = Data byte has been transmitted, ACK has been received.
;MASTER TRANSMITTER MODE.
;This section is entered when a data byte has been successfully transmitted.
;Now the system has to check if more data bytes are to be sent, and if not,
;should a subroutine be called before going on to next IIC block in the
;IIC command file.
;-----
MORE_28:
MOV      Attempt_count,#0          ;clear attempt count since all OK
MOV      A,Multiple_count         ;check for end of data bytes out
JZ       M28_03                   ;last byte has been sent
DEC      Multiple_count           ;more bytes to be sent so decrement
JMP      M18_20                   ;count and send next byte
;
;enter here when all data bytes sent.
M28_03:
JBC      call_function,M28_20     ;check for request to call subroutine
M28_05:
CALL     FETCH_COMMAND_0          ;check for end-of-session
CJNE     A,#iicend,M28_10
M28_X:
MOV      IIC_status,#status_OK
JMP      M00_10
;
;if not end-of-session, do another start
M28_10:
MOV      S1CON,#ENSI_STA_NOTSTO_NOTSI_AA_CR0
JMP      IIC_EXIT
;
;enter here if a 'call_' to a subroutine is requested. First push
;the return address (above) onto stack, then get the address of the
;subroutine to call from the IIC command file. Push the call address
;onto the stack (low address first), then call subroutine.
;
;The called subroutine could be used to modify the contents of the
;'IIC_Command_File_adrs' registers. In doing so, IF-THEN-ELSE
;control flow could be done (i.e. based on some IIC read information,
;the subroutine may decide to run one of several other IIC blocks,
;or end the session altogether). More likely, the subroutine will be
;used to manipulate some data before it is transmitted.
M28_20:
MOV      A,#LOW(M28_05)           ;put return address onto stack
PUSH     ACC
MOV      A,#HIGH(M28_05)
PUSH     ACC
CALL     FETCH_COMMAND            ;get address of subroutine from command file
PUSH     ACC                      ;and put it onto the stack (LSB first)
CALL     FETCH_COMMAND
PUSH     ACC
RET                                  ;CALL the subroutine

```


Multimaster I²C routines for byte-oriented I²C interfaces AN435

```

;State 30H = Data byte has been transmitted, NO ACK received.
;MASTER TRANSMITTER MODE.
;This state is similar to state 20H, except that data has been transmitted,
;not an address.
;The routine 'FETCH_DATA' always stores the data fetched as 'last_data' so
;that in the case of a NO ACK, it can be re-transmitted.
;-----
;
;test-up to return a NOT ACK on next data reception.
;
MORE_30:
    INC     Attempt_count           ;bump attempt count
    MOV     A,Attempt_count         ;if too many attempt, error
    CJNE    A,#max_data_attempts,M30_10
    MOV     IIC_status,#status_attempt_data ;error status update
    JMP     M00_10

M30_10:
    MOV     A,last_data             ;get data not received and
    JMP     M18_25                 ;re-send it

;
;-----
;State 38H = Arbitration lost to another master.
;MASTER TRANSMITTER MODE.
;If this state is entered, simply let the other Master have the run of the
;bus. The mainline routine that started the IIC session can check
;the 'IIC_status' register for this state and re-try later.
;-----
;
;Multiple_count
;IIC_EXIT
MORE_38:
    MOV     IIC_status,#status_arb_lost ;error status update
    JMP     M40_20

;
;-----
;State 40H = (Slave Address + Read bit) has been transmitted, ACK received.
;MASTER RECEIVER MODE.
;This state is entered when the byte must be stored, then a check must be done
;for the calling of a subroutine with that state.
;See State 18H for more details.
;-----
;
;-----
;
MORE_40:
    MOV     Attempt_count,#0
M40_15:
    CALL    FETCH_COMMAND
    CALL    make_space
    CALL    M18_SUB

M40_19:
    MOV     A,Multiple_count
    JNZ     M40_20
    MOV     S1CON,#ENSI_NOTSTA_NOTSTO_NOTSI_NOTAA_CR0
    JMP     IIC_EXIT

M40_20:
    MOV     S1CON,#ENSI_NOTSTA_NOTSTO_NOTSI_AA_CR0
    JMP     IIC_EXIT

;
;-----
;State 48H = (Slave address + Read bit) transmitted, NOT ACK received.
;MASTER RECEIVER MODE.
;See State 20H.
;-----
;
;-----
;
MORE_48:
    JMP     MORE_20

```

```

;-----
;State 50H = Data byte has been received, ACK returned.
;MASTER RECEIVER MODE.
;This state stores the received data byte and determines whether more data is
;required or not. If more data required (i.e. 'Multiple_count' > 1), then
;send back an ACK, if no more data to be received ('Multiple_count' = 1), then
;set-up to return a NOT ACK on next data byte reception.
;-----
MORE_50:
MOV A,S1DAT ;get received data byte
CALL STORE_DATA ;store the data in appropriate space
MOV Attempt_count,#0 ;indicate all OK
MOV A,Multiple_count ;check for more bytes to be received
CJNE A,#1,M50_10
;
;If next byte to be received is last, make sure a NOT ACK is sent
;with next reception.
;-----
M50_05:
MOV S1CON,#ENSI_NOTSTA_NOTSTO_NOTSI_NOTAA_CRO
SJMP M50_15
M50_10:
MOV S1CON,#ENSI_NOTSTA_NOTSTO_NOTSI_AA_CRO
M50_15:
DEC Multiple_count
JMP IIC_EXIT
;-----
;State 58H = Data byte has been received, NOT ACK has been returned.
;MASTER RECEIVER MODE.
;This state is entered when the last byte required has been received by the
;Master. In this case, the byte must be stored, then a check must be done
;for the calling of a subroutine, and/or the end of the entire IIC session.
;See State 28H for more details.
;-----
MORE_58:
MOV A,S1DAT ;get received byte
CALL STORE_DATA ;store it
MOV Attempt_count,#0 ;clear error flag
JMP M28_03 ;check for end-of-session or 'call_'
;-----
;State 60H = Own Slave Address (+ Write bit) has been received,
;ACK has been returned.
;SLAVE RECEIVER MODE.
;When own address found, this system will receive 'SLVbytes_in' bytes of data
;into 'Slave_in' data space.
;The calling master must produce the stop or repeated start conditions. This
;micro was not in an active IIC mode when the other master addressed it, so
;the "WAIT_IIC" subroutine is not active, thus timeout problems will not be
;checked for unless "WAIT_IIC" is called. "WAIT_IIC" will do only a timeout
;check if called from the main program since it will wait for the 'IIC_status'
;to become 'status_OK'.
;-----
MORE_60:
MOV IIC_status,#status_slave
M60_10:

```

Multimaster I²C routines for byte-oriented I²C interfaces AN435

```

SETB     i_am_a_slave
MOV      Multiple_count, #(SLVbytes_in - 1)    ;set for # bytes received
MOV      Command?adrs?space, #ioD             ;receive bytes into DATA space
MOV      IO_buffer_adrs, #Slave_in            ;address of DATA space target
SJMP     M40_20

;
;-----
;State 68H = Arbitration lost while addressing a slave; Own slave address and
; write bit has been received.
;SLAVE RECEIVER MODE.
;Indicate that arbitration is lost so that the "WAIT_IIC" routine is aborted
;and the interrupt from the IIC hardware runs the system.
; "WAIT_IIC" is active if this state is entered since state 68H is entered
; upon lost arbitration for the bus.
; "WAIT_IIC" will terminate in this case since the 'IIC_status' will show that
; another master has won the bus.
;-----
;
;
MORE_68:
MOV      IIC_status, #status_arb_lost_slave
SJMP     M60_10

;
;-----
;State 70H = General call address (00H) has been received, ACK has been
; returned (by this micro).
;SLAVE RECEIVER MODE.
;Indicates that a general call has been received - 'SLVbytes_in' bytes will
; be received into 'Slave_in' as if this slave were addressed.
;-----
;
;
MORE_70:
MOV      IIC_status, #status_general_slave
SJMP     M60_10

;
;-----
;State 78H = Arbitration lost while addressing a slave - General call address
; (00H) has been received, ACK has been returned (by this micro).
;SLAVE RECEIVER MODE.
;Indicates that a general call has been received - 'SLVbytes_in' bytes will be
; received into 'Slave_in' as if this slave were addressed.
;-----
;
;
MORE_78:
MOV      IIC_status, #status_arb_lost_general
SJMP     M60_10

;
;-----
;State 80H = Previously addressed with own slave address; data has been
; received, ACK has been returned (by this micro).
;SLAVE RECEIVER MODE.
;Data byte received in 'S1DAT', ACK returned.
;-----
;
;
MORE_80:
SJMP     MORE_50

;
;-----
;State 88H = Previously addressed with own slave address; data byte has been
; received, NOT ACK has been returned (by this micro).
;SLAVE RECEIVER MODE.
;Last byte to be received is in 'S1DAT'. A NACK has been returned.
;-----

```

Multimaster I²C routines for byte-oriented I²C interfaces AN435

```

; Slave Receiver Mode.
M88_80: MOV A,S1DAT ;get received byte
CALL STORE_DATA ;store it

M88_10: MOV IIC_status,#status_OK
CLR i_am_a_slave
SJMP M40_20

;-----
;State 90H = Previously addressed with general call; data byte has been
; received, ACK has been returned (by this micro).
SLAVE RECEIVER MODE.

;-----
MORE_90: SJMP MORE_80

;-----
;State 98H = Previously addressed with general call; data byte has been
; received, NOT ACK has been returned (by this micro).
SLAVE RECEIVER MODE.

;-----
MORE_98: SJMP MORE_88

;-----
;State A0H = a STOP or repeated START has been received while still in the
; addressed slave receiver or transmitter mode.
SLAVE RECEIVER MODE.

;-----
MORE_A0: SJMP M88_10

;-----
;State A8H = Own slave address + read byte has been received; ACK has been
; returned (by this micro).
SLAVE TRANSMITTER MODE.
;This micro has been addressed by another master, and has been told to send
;data. This micro will respond by sending 'SLVbytes_out' bytes of data from
;'Slave_out'.

;-----
MORE_A8: MOV IIC_status,#status_slaves_read
MA8_10: SETB i_am_a_slave
MOV Multiple_count,#(SLVbytes_out) ;set for 2 bytes to be sent
MOV Command?adrs?space,#ioD ;transmit bytes from DATA space
MOV IO_buffer_adrs,#Slave_out ;address of DATA space target
MOV IO_buffer_adrs+1,#0
SJMP MORE_B8

;-----
;State B0H = Arbitration lost while trying to get to a slave; own slave
; address + read has been received; ACK has been returned (by this
; micro).
SLAVE TRANSMITTER MODE.

```

Multimaster I²C routines for byte-oriented I²C interfaces AN435

```

;-----
;
;
MORE_B0:
    MOV     IIC_status,#status_arb_lost_slave
    SJMP    M88_10
;
;-----
;State B8H = Data byte in S1DAT has been transmitted; ACK has been received.
;SLAVE TRANSMITTER MODE.
;This section checks if any more bytes are to be transmitted.
;-----
;
;
MORE_B8:
    MOV     A,Multiple_count
    JZ      M88_03
    DEC     Multiple_count
MB8_03:
    CALL    FETCH_DATA ;now ready to get data byte
    MOV     S1DAT,A ;send as data
    JMP     M40_19
;
;-----
;State C0H = Data byte in S1DAT has been transmitted; NOT ACK has been
; received.
;SLAVE TRANSMITTER MODE.
;This is the end of the addressed slave session. A STOP or repeated START
;will be the next state, but this addressed slave doesn't care unless the next
;address sent by the calling master is it's own, or the general call address.
;-----
;
MORE_C0:
    JMP     M88_10
;
;-----
;State C8H = Last data byte in S1DAT has been transmitted; ACK has been
; received.
;SLAVE TRANSMITTER MODE.
;Treated same as state C0.
;-----
;
MORE_C8:
    JMP     M88_10

CODE_start EQU $
}

```


Summary:

This application note presents a set of software routines, to drive the I²C interface in 8xC528 type of micro controllers. A description of the I²C interface is given. Examples show how to use these routines in PL/M-51, C and assembly source code.

1. INTRODUCTION

This application note describes the I²C interface of the 8xC528 μ C and gives a set of routines in application programs to drive this interface.

Chapter 2 gives a hardware description of the bit level I²C. It gives an overview what functions are done in hardware by the interface and the functions that should be implemented by software. The registers described are accessible with software and control the I²C interface.

Chapter 3 gives a description of the routines that may be used by the application program. The routines are written in such a way that the I²C interface becomes transparent to the user. The slave program is descibed in more detail, because this routine may be adapted by the user for his specific application.

Chapter 4 gives simple example programs that show how to use the routines in assembly, PL/M and C application programs.

References:

- The I²C-bus specification; 9398 358 10011
- 8051-based 8-bit Microcontrollers; Data Handbook IC20
- PLM51 I²C Software interface IIC51; ETV/AN89004

I²C routines for 8XC528

AN438

2. THE I²C INTERFACE

2.1 CHARACTERISTICS OF I²C INTERFACE

On page 4 the block diagram of the bit-level I²C interface is shown. P1.6/SCL and P1.7/SDA are the serial I/O pins. These two pins meet the I²C specification concerning the input levels and output drive capability. Consequently, these pins have an open drain configuration. All four modes of the I²C bus can be used:

- Master transmitter
- Master receiver
- Slave transmitter
- Slave receiver

The advantages of using the bit-level I²C hardware compared with a full software implementation are:

- Higher bit rate
- No critical software timing requirements
- Less software overhead
- More reliable data transfer

The bit-level I²C hardware can perform the following functions:

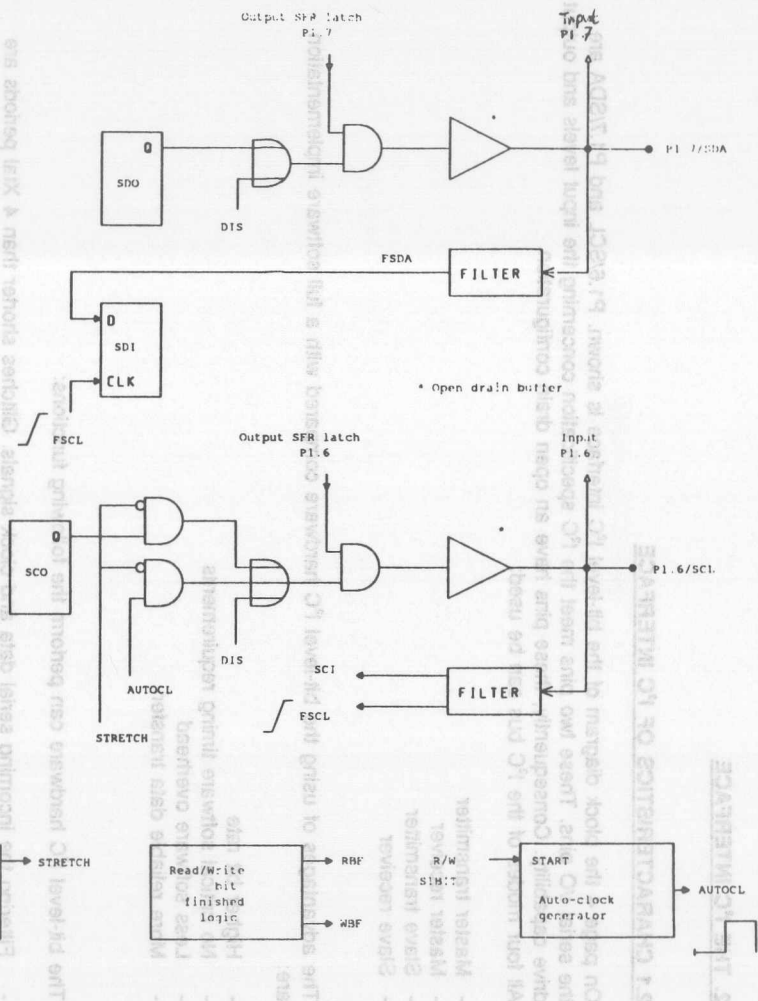
- Filtering the incoming serial data and clock signals. Glitches shorter than 4 Xtal periods are rejected.
- Recognition of a START or STOP condition.
- Generating an interrupt request after reception of a START condition.
- Setting the Bus Busy flag when a START condition is detected.
- Clearing the Bus Busy flag when a STOP condition is detected.
- Recognition of a serial clock pulse on the SCL line.
- Latching the serial data bit on the SDA line at every rising edge on the SCL line.
- Stretching the LOW period of the serial clock SCL to synchronize with external master devices.
- Setting the Read Bit Finished (RBF) or Write Bit Finished (WBF) flag if an error free bit transfer has occurred.
- Setting a Clock LOW-to-HIGH (CLH) flag when a leading edge is detected on the SCL line.
- Generation of serial clock pulse on SCL in master mode.

The following functions must be done with software:

- Handling the I²C interrupt caused by a detected START condition.
- Conversion of serial to parallel data when receiving.
- Conversion of parallel to serial data when transmitting.
- Comparing received slave address with own slave address.
- Interpretation of acknowledge information.
- Guarding the I²C status if the RBF and WBF flags indicate a not regular bit transfer.
- Generating START/STOP conditions when in master mode.
- Handling bus arbitration when in master mode.

I²C routines for 8XC528

82C0XB to 82C0XA AN438

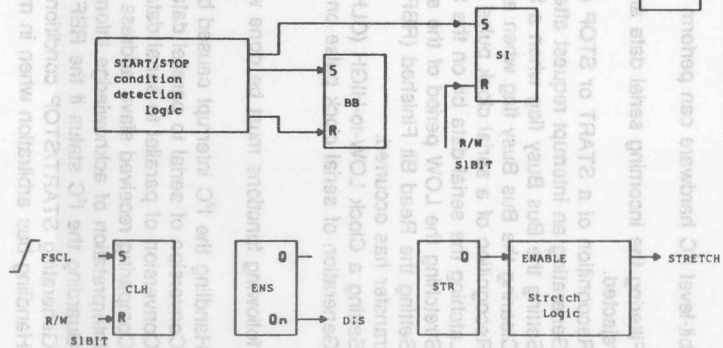


Special Function Registers

SIINT (DAh)	7	6	5	4	3	2	1	0	R/W
	SI	X	X	M	X	X	X	X	
SIBIT (D9h)	7	6	5	4	3	2	1	0	R
	SDI	0	0	0	0	0	0	0	
	SDO	X	X	X	X	X	X	X	W
Bit address	DF	DE	DD	DC	DB	DA	D9	D8	
SISCS (DBh)	7	6	5	4	3	2	1	0	R
	SDI	SCI	CLH	BB	RBF	WBF	STR	ENS	
	SDO	SCO	CLH	X	X	X	STR	ENS	W

- Software can only clear this bit
- This bit is read with read-modify-write operation

X = Undefined (R) or don't care (W)
R = Read access
W = Write access



I²C routines for 8XC528

AN438

2.2 CONTROL AND STATUS REGISTERS

Control of the I²C bus hardware is done via 3 Special Function Registers:

- S1INT: This register contains the serial interrupt flag SI.
- S1BIT: For read this register contains the received bit SDI.
For write this register contains bit SDO to be transmitted.
- S1SCS: For read this register contains status information.
For write this register is used as control register.

2.2.1 S1INT: I²C interrupt register

- **S1INT.7** is the Serial Interrupt request flag (SI).
If the serial I/O is enabled (ENS = 1), then a START condition will be detected and the SI flag is set on the falling edge of the filtered SCL signal.
Provided that EA (global enable) and ES1 (enable I²C interrupt) are set (in the interrupt enable IE register), SI generates an interrupt that will start the slave address receive routine.
SI is cleared by accessing the S1BIT register or by writing '00H' to S1INT. SI cannot be set by software

After reception of a START condition, the LOW period of the SCL pulse is stretched, suspending serial transfer to allow the software to take appropriate action. This clock stretching is ended by accessing the S1BIT register.

2.2.2 S1BIT: Single bit data register

- **S1BIT.7** contains two physical latches; the Serial Data Output (SDO) latch for a write operation and the filtered Serial Data Input (SDI) latch for a read operation. SDI data is latched on the rising edge of the filtered SCL pulse. S1BIT.7 accesses the same physical latches as S1SCS.7, but S1BIT.7 is not bit-addressable.

Reading or writing S1BIT register starts the next additional actions:

- SI, CLH, RBF and WBF flags are cleared
- Stretching the LOW period of the SCL clock is finished.
- Auto-clock pulse is started if enabled

The auto-clock is an active HIGH SCL pulse that starts 28 Xtal periods after an access to S1BIT. SCL remains high for 100 Xtal periods. If the SCL line is kept LOW by any device that wants to hold up the bus transfer, the auto-clock counter still runs for 20 Xtal periods to try to make SCL high and then go in a wait-state. This will result in a minimum SCL HIGH time of 80 Xtal periods (5µs at $f_{xtal} = 16\text{MHz}$).

The auto-clock signal will be inhibited if the SCO flag in the S1SCS register is set to '1'. SCL pulses must then be generated by software. In this situation access to S1BIT may be used to clear the SI, CLH, RBF and WBF flags.

A quick check on a successful bit transfer from/to SDO/SDI is carried out by testing only the RBF or WBF flag (see 2.2.3).

I²C routines for 8XC528

AN438

2.2.3 S1SCS: Control and status register

- **S1SCS.7** represents two physical latches, the Serial Data Output (SDO) latch for write operations and the Serial Data Input (SDI) latch for read operations. S1SCS.7 accesses the same physical latches as S1BIT.7, but S1SCS.7 is bit addressable. However a read or write operation of S1SCS.7 does not start an auto-clock pulse, will not finish clock stretching and will not clear flags!
- **S1SCS.6** represents two physical latches, the Serial Clock Output (SCO) latch for write operations and the Serial Clock Input (SCI) latch for read operations. The output of SCO is "OR-ed" with the auto-clock pulse. If SCO = '1' the auto-clock generation is disabled and its output is LOW. Internal clock stretching logic and external devices then can pull the SCL line LOW. If the auto-clock is not used the SCL line has to be controlled by setting SCO = '1', waiting for CLH to become '1' and setting SCO = '0' after the specified SCL HIGH time. Data access should be done via S1SCS.7.
- **S1SCS.5** is the serial Clock LOW-to-HIGH transition flag (CLH). This flag is set by a rising edge of the filtered serial clock. CLH = '1' indicates that no devices are stretching SCL LOW, and since the last CLH reset, a new valid data bit has been latched in SDI. CLH can be cleared by writing '0' to S1SCS.5 or by a read or write operation to the S1BIT register. Clearing CLH also clears RBF and WBF. Writing a '1' to S1SCS.5 will not affect CLH.
- **S1SCS.4** is the Bus Busy flag (BB). BB is set or cleared by hardware only. If set it indicates that a START condition has been detected on the I²C bus. A STOP condition clears the BB-flag.
- **S1SCS.3** is the Read Bit Finished flag (RBF). If RBF = 1 it indicates that a serial bit has been received and latched into SDI successfully. If during a bit transfer RBF is '0', the cause is indicated as follows:
 - SCI = '1' and CLH = '1': The SCL pulse is not finished and still HIGH.
 - CLH = '0': A bus device is delaying the transfer by stretching the LOW level on the SCL line.
 - BB = '0': A STOP-condition has been detected during the bit transfer. This should be considered as a bus-error.
 - SI = '1': A START-condition has been detected during the bit transfer. This should be considered as a bus-error.
 RBF can be cleared by clearing CLH or by a read or write operation to the S1BIT register.
- **S1SCS.2** is the Write Bit Finished flag (WBF). If set it indicates that a serial bit in SDO has been transmitted successfully. If during bit transfer WBF is '0', the following conditions may be the cause:
 - SCI = '1' and CLH = '1': The SCL pulse is not finished and still HIGH.
 - CLH = '0': A bus device is delaying the transfer by stretching the LOW level on the SCL line.
 - BB = '0': A STOP-condition has been detected during the bit transfer. This should be considered as a bus-error.
 - SI = '1': A START-condition has been detected during the bit transfer. This should be considered as a bus-error.
 WBF can be cleared by clearing CLH or access to the S1BIT register.

I²C routines for 8XC528

AN438

- **S1SCS.1** is the STRetch control flag (STR). STR can be set or cleared by software only. Setting STR enables the stretching of SCL LOW periods. Stretching will occur after a falling edge on the filtered serial clock. This allows synchronization with the SCL clock signal of an external master device.
If STR is cleared, no stretching of the SCL LOW period will occur after the transfer of a serial bit.
The LOW level on the SCL line is also stretched after a START condition is received, regardless of the STR contents. The stretching of the SCL LOW period is finished by a read or write operation of the S1BIT register.
- **S1SCS.0** is the ENable Serial I/O flag (ENS). ENS can be set or cleared by software only.
ENS = '0' disables the serial I/O. The I/O signals P1.6/SCL and P1.7/SDA are determined by the port latches of P1.6 and P1.7 (open drain). If P1.6 and P1.7 are connected to an I²C bus, then the flags SDI, SCI, CLH and BB still monitor the I²C bus status, but will not influence the I/O lines, nor will they request an interrupt.
ENS='1' enables the START detection and clock stretching logic. Note that the P1.6 and P1.7 latches and the SDO and SCO control flags must be set to '1' before ENS is set to avoid SCL and/or SDA to pull the lines LOW.

3: I²C ROUTINES

3.1 INTRODUCTION

A set of routines is written for the I²C interface that supports multi-master and slave operation. The routines are placed in a library I2C_DR.LIB. If I2C_DR.LIB is linked to an application program, only the needed object modules are linked in the output file.
The routines can be used as device driver for PL/M-51, C and 8051-assembly code. By using these routines the bit-level I²C interface is fully transparent for the user.

The routines use the following 8xC528 resources:

- Exclusive use of Register_Bank_1. Only R7 of this register bank contains static data (Own Slave Address). R0..R6 may be used by the application program when the I²C routine is finished.
- 7 bytes DATA used for parameter passing.
- 1 byte Bit-Addressable DATA for status flags.

When using routines from this library DPH, DPL, PSW (except CY) and B are not altered.

An n-bytes data buffer is used as destination or source buffer for the bytes to be received/transmitted and reside in DATA or IDATA memory space.

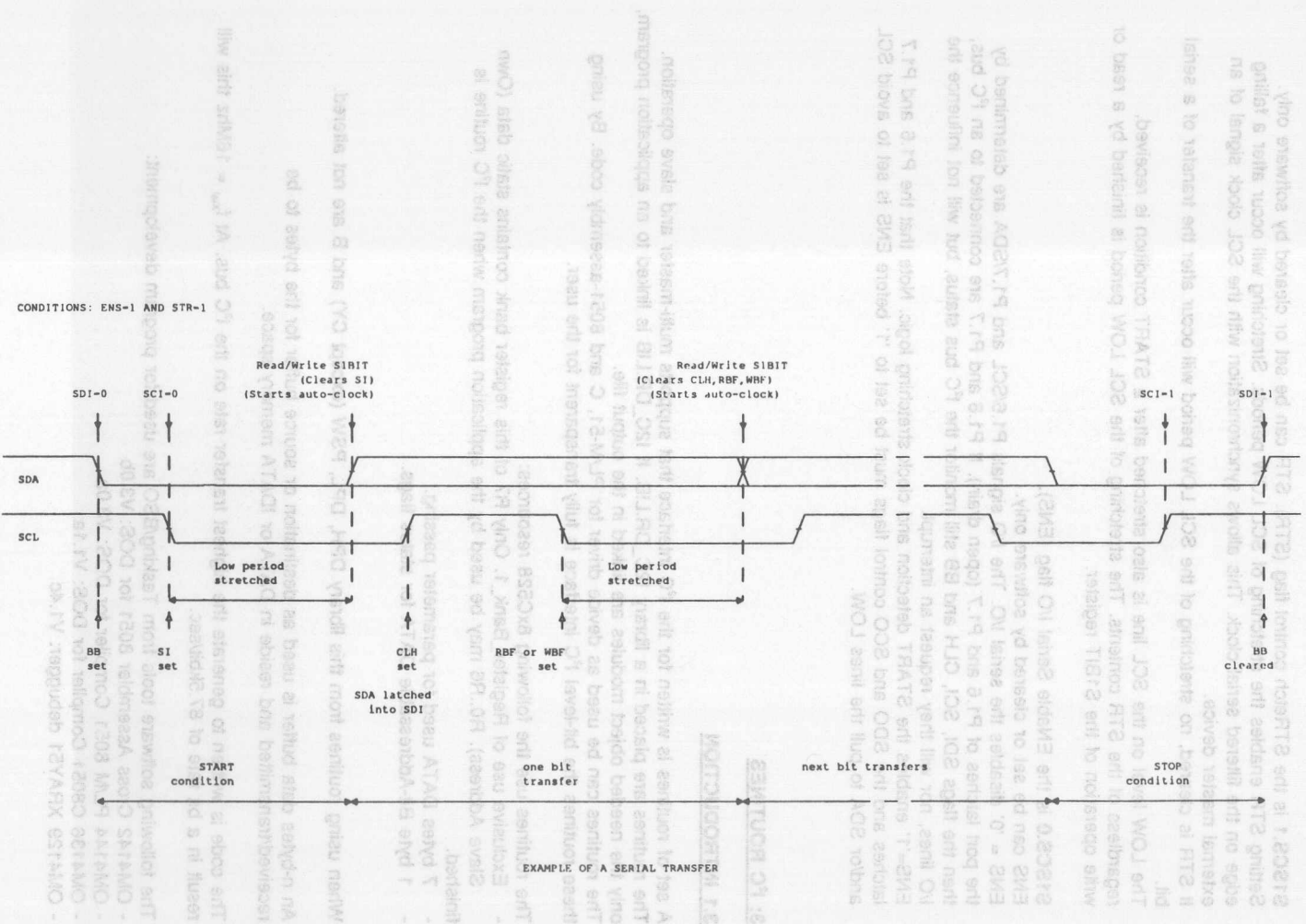
The code is written to generate the highest transfer rate on the I²C bus. At $f_{\text{xtal}} = 16\text{Mhz}$ this will result in a bit rate of 87.5kbit/sec.

The following software tools from Tasking/BSO are used for program development:

- OM4142 Cross Assembler 8051 for DOS: V3.0b
- OM4144 PL/M 8051 Compiler for DOS: V3.0a
- OM4136 C8051 Compiler for DOS: V1.1a
- OM4129 XRAY51 debugger: V1.4c

^{12}C routines for 8XC528

AN438



I²C routines for 8XC528

AN438

3.2 FUNCTIONAL DESCRIPTION

When using these routines in a PL/M application program they must be declared EXTERNAL. In this declaration the user can specify the type returned by each procedure. All procedures (except Init_IIC and Dis_IIC) can return a BIT or BYTE, depending on the chosen EXTERNAL declaration. The BIT or BYTE returned is '0' if the I²C was successful. If a BYTE is returned the following check bits are available for the user:

BYTE.0: An I²C error has been detected.

BYTE.1: No ACK received.

BYTE.2: Arbitration lost.

BYTE.3: Time out error. This may be caused by an external device pulling SCL LOW.

BYTE.4: A bus error has occurred. This may be a spurious START/STOP during a bit transfer.

BYTE.5: No access to I²C bus.

BYTE.6: 0

BYTE.7: 0

Note that typed procedures must be called using an expression. If the result of an I²C procedure is to be ignored, a dummy assignment must be done for a typed procedure. The examples in the following section assume that the procedures are called from a PL/M program. Examples will be given later how to use these routines with C and assembly application programs.

3.2.1 Init_IIC

Declaration:

Init_IIC:

```
PROCEDURE (Own_Slave_Address, Slave_Sub_Address) EXTERNAL;
DECLARE (Own_Slave_Address, Slave_Sub_Address) BYTE;
END;
```

Description:

Init_IIC must be called after RESET, before any procedure is called. The I²C interface and I²C interrupt will be enabled. The global enable interrupt flag however will not be effected. This should be done afterwards. Own_Slave_Address is passed to Init_IIC for use as slave. Slave_Sub_Address is the pointer to a DATA buffer that is used for data transfer in slave mode. When used as master in a single master system, these parameters are not used.

Example:

```
CALL Init_IIC (54h,.Slave_Data_Buffer);
ENABLE; /* Enable Interrupts; EA=1 */
```

3.2.2 Dis_IIC

3.2 FUNCTIONAL DESCRIPTION

Declaration:

Dis_IIC:
PROCEDURE EXTERNAL;

Description:

Dis_IIC will disable the I²C-interface and the I²C-interrupt.
The I²C interface will still monitor the bus, but will not influence the SDA and SCL lines.

Example:

CALL Dis_IIC;

3.2.3 IIC_Test_Device

Declaration:

IIC_Test_Device:
PROCEDURE (Slave_Address) [BIT|BYTE] EXTERNAL;
DECLARE (Slave_Address) BYTE;
END;

Description:

IIC_Test_Device just sends the slave address to the I²C bus. It can be used to check the presence of a device on the I²C bus

I²C Protocol:

S-SlW-A-P : Device is present, IIC_Error=0
S-SlW-N-P : Device is not present, IIC_Error=1

Example:

DECLARE IIC_Error BIT;
.....
IIC_Error=IIC_Test_Device(8Ch);
IF (IIC_Error) THEN
 "Device not acknowledging on slave address"
ELSE
 "Device acknowledges on slave address"

I²C routines for 8XC528

AN438

3.2.4 IIC_Write

Declaration:

IIC_Write:

```

PROCEDURE (Slave_Address, Count, Source_Ptr) [BIT|BYTE] EXTERNAL;
DECLARE (Slave_Address, Count, Source_Ptr) BYTE;
END;
```

Description:

IIC_Write is the most basic procedure to write a message to a slave device.

I²C Protocol:

```

L           =Count
D1[0..L-1]  BASED by Source_Ptr

S-SlvW-A-D1[0]-A....A-D1[L-1]-A-P
```

Example:

```

DECLARE Data_Buffer(4) BYTE;
.....
CALL IIC_Write(02Ch, LENGTH(Data_Buffer),..Data_Buffer);
```


I²C routines for 8XC528

AN438

3.2.5 IIC_Write_Sub

Declaration:

IIC_Write_Sub:

```

PROCEDURE (Slave_Address, Count, Source_Ptr, Sub_Address) [BIT|BYTE] EXTERNAL;
DECLARE (Slave_Address, Count, Source_Ptr, Sub_Address) BYTE;
END;
```

Description:

IIC_Write_Sub writes a message preceded by a sub-address to a slave device

I²C Protocol:

```

L           =Count
Sub         =Sub_Address
D1[0..L-1]  BASED by Source_Ptr
```

S-SlvW-A-Sub-A-D1[0]-A-D1[1]-A.....A-D1[L-1]-A-P

Example:

```

DECLARE Data_Buffer(8) BYTE;
.....
CALL IIC_Write_Sub (48h,LENGTH(Data_Buffer),Data_Buffer,2);
```

I²C routines for 8XC528

AN438

3.2.6 IIC_Write_Sub_SWInc

Declaration:

IIC_Write_Sub_SWInc:

```

PROCEDURE (Slave_Address, Count, Source_Ptr, Sub_Address) [BIT|BYTE] EXTERNAL;
DECLARE (Slave_Address, Count, Source_Ptr, Sub_Address) BYTE;
END;
```

Description:

Some I²C devices addressed with a sub-address do not automatically increment the sub-address after reception of each byte. IIC_Write_Sub_SWInc can be used for such devices the same way as IIC_Write_Sub is used. IIC_Write_Sub_SWInc splits up the message in smaller messages and increments the sub-address itself.

I²C Protocol:

```

L          =Count
Sub        =Sub_Address
D1[0..L-1] BASED by Source_Ptr

S-SlvW-A- (Sub+0) - A-D1[0] - A-P
S-SlvW-A- (Sub+1) - A-D1[1] - A-P
.....
S-SlvW-A- (Sub+L-1)-A-D1[L-1]-A-P
```

Example:

```

DECLARE Data_Buffer(6) BYTE;
.....
CALL IIC_Write_Sub_SWInc(80h,LENGTH(Data_Buffer),Data_Buffer,2);
```

I²C routines for 8XC528

AN438

3.2.7 IIC_Write_Memory

Declaration:

IIC_Write_Memory:

```
PROCEDURE (Slave_Address, Count, Source_Ptr, Sub_Address) [BIT|BYTE] EXTERNAL;
DECLARE (Slave_Address, Count, Source_Ptr, Sub_Address) BYTE;
END;
```

Description:

I²C Non-Volatile Memory devices (such as PCF8582) need an additional delay after writing a byte to it. IIC_Write_Memory can be used to write to such devices the same way IIC_Write_Sub is used. IIC_Write_Memory splits up the message in smaller messages and increments the sub-address itself. After transmission of each message a delay of 40 milliseconds ($f_{\text{xtal}} = 16\text{MHz}$) is inserted.

I²C Protocol:

```
L          =Count
Sub        =Sub_Address
D1[0..L-1] BASED by Source_Ptr

S-SlvW-A- (Sub+0) - A- D1[0] - A-P
Delay 40ms
S-SlvW-A- (Sub+1) - A- D1[1] - A-P
Delay 40ms
.....
S-SlvW-A- (Sub+L-1)- A- D1[L-1]-A-P
Delay 40ms
```

Example:

```
DECLARE Data_Buffer(10) BYTE;
.....
CALL IIC_Write_Memory(0A0h,LENGTH(Data_Buffer),Data_Buffer,0F0h);
```

I²C routines for 8XC528

AN438

3.2.8 IIC_Write_Sub_Write

Declaration:

IIC_Write_Sub_Write:

PROCEDURE (Slave_Address, Count1, Source_Ptr1, Sub_Address, Count2, Source_Ptr2)

[BIT|BYTE] EXTERNAL;

DECLARE (Slave_Address, Count1, Source_Ptr1, Sub_Address, Count2, Source_Ptr2)

BYTE;

END;

Description:

IIC_Write_Sub_Write writes 2 data blocks preceded by a sub-address in one message to a slave device. This procedure can be used for devices that need an extended addressing method, without the need to put all data into one large buffer. Such a device is the ECCT (I²C controlled teletext device; see example).

I²C Protocol:

L =Count1

M =Count2

Sub =Sub_Address

D1[0..L-1] BASED by Source_Ptr1

D2[0..M-1] BASED by Source_Ptr2

S-SlvW-A-Sub-A-D1[0]-A-D1[1]-A-.....-A-D1[L-1]-A-D2[0]-A-D2[1]-A-.....-A-D2[M-1]-A-P

Example:

PROCEDURE Write_CCT_Memory (Chapter, Row, Column, Data_Buf, Data_Count);

DECLARE (Chapter, Row, Column, Data_Buf, Data_Count) BYTE;

/* The extended address (CCT-Cursor) is formed by Chapter, Row and Column. These three bytes are written after the sub-address (=8) followed by the actual data that will be stored relative to the extended address. */

CALL IIC_Write_Sub_Write (22h, 3, .Chapter, 8, Data_Buf, Data_Count);

END Write_CCT_Memory;

I²C routines for 8XC528

AN438

3.2.9 IIC_Write_Sub_Read

Declaration:

IIC_Write_Sub_Read:

```

PROCEDURE (Slave_Address, Count1, Source_Ptr1, Sub_Address, Count2, Dest_Ptr2)
    [BIT|BYTE] EXTERNAL;
DECLARE (Slave_Address, Count1, Source_Ptr1, Sub_Address, Count2, Dest_Ptr2)
    BYTE;
END;
```

Description:

IIC_Write_Sub_Read writes a data block preceded by a sub-address, generates an I²C restart condition, and reads a data block. This procedure can be used for devices that need an extended addressing method. Such a device is the ECCT.

I²C Protocol:

```

L      =Count1
M      =Count2
Sub    =Sub_Address
D1[0..L-1] BASED by Source_Ptr1
D2[0..M-1] BASED by Source_Ptr2
```

S-SlvW-A-Sub-A-D1[0]-A-D1[1]-A-.....-A-D1[L-1]-A-S-SlvR-A-D2[0]-A-D2[1]-A-.....-A-D2[M-1]-N-P

Example:

```

PROCEDURE Read_CCT_Memory (Chapter, Row, Column, Data_Buf, Data_Count);
DECLARE (Chapter, Row, Column, Data_Buf, Data_Count) BYTE;
```

/*

The extended address (CCT-Cursor) is formed by Chapter, Row and Column. These three bytes are written after the sub-address (8). After that the actual data will be read relative to the extended address.

*/

```

CALL IIC_Write_Sub_Write (22h, 3, Chapter, 8, Data_Buf, Data_Count);
END Read_CCT_Memory;
```


I²C routines for 8XC528

AN438

3.2.10 IIC_Write_Com_Write

Declaration:

IIC_Write_Com_Write:

```
PROCEDURE (Slave_Address, Count1, Source_Ptr1, Count2, Source_Ptr2) [BIT|BYTE]
```

EXTERNAL;

```
DECLARE (Slave_Address, Count1, Source_Ptr1, Count2, Source_ptr2) BYTE;
```

```
END;
```

Description:

IIC_Write_Com_Write writes two data blocks from different data buffers in one message to a slave receiver. This procedure can be used for devices where the message consists of 2 different data blocks. Such devices are for instance LCD-drivers, where the first part of the message consists of addressing and control information, and the second part is the data string to be displayed.

I²C Protocol:

L =Count1

M =Count2

D1[0..L-1] BASED by Source_Ptr1

D2[0..M-1] BASED by Source_Ptr2

S-SlvW-A-D1[0]-A-D1[1]-A-....-A-D1[L-1]-A-D2[0]-A-D2[1]-A-....-A-D2[M-1]-A-P

Example:

```
DECLARE Control_Buffer(2) BYTE;
```

```
DECLARE Data_Buffer(20) BYTE;
```

```
.....
```

```
CALL IIC_Write_Com_Write(74h, LENGTH(Control_Buffer), .Control_Buffer,  
LENGTH(Data_Buffer), .Data_Buffer);
```

I²C routines for 8XC528

AN438

3.2.11 IIC_Write_Rep_Write

Declaration:

IIC_Write_Rep_Write:

```
PROCEDURE (Slave_Address1, Count1, Source_Ptr1, Slave_Address2, Count2, Source_ptr2)
```

```
[BIT|BYTE] EXTERNAL;
```

```
DECLARE (Slave_Address1, Count1, Source_ptr1, Slave_Address2, Count2, Source_Ptr2)
```

```
BYTE;
```

```
END;
```

Description:

Two data strings are sent to separate slave devices, separated with a repeat START condition. This has the advantage that the bus does not have to be released with a STOP condition before the transfer from the second slave.

I²C Protocol:

```
L      =Count1
```

```
M      =Count2
```

```
SlvW1   =Slave_Address1
```

```
SlvW2   =Slave_Address2
```

```
D1[0..L-1] BASED by Source_Ptr1
```

```
D2[0..M-1] BASED by Source_Ptr2
```

```
S-SlvW-A-D1[0]-A-D1[1]-.....-A-D1[L-1]-A-S-SlvW-A-D2[0]-A-D2[1]-.....-A-D2[M-1]-A-P
```

Example:

```
DECLARE Data_Buffer_1(10) BYTE;
```

```
DECLARE Data_Buffer_2(4) BYTE;
```

```
.....
```

```
CALL IIC_Write_Rep_Write (48h, LENGTH(Data_Buffer_1), .Data_Buffer_1, 50h,  
                           LENGTH(Data_Buffer_2), .Data_Buffer_2);
```

I²C routines for 8XC528

AN438

3.2.12 IIC_Write_Rep_Read

Declaration:

IIC_Write_Rep_Read:

```

PROCEDURE (Slave_Address1, Count1, Source_Ptr1, Slave_Address2, Count2, Dest_ptr2)
  [BIT|BYTE] EXTERNAL;
DECLARE (Slave_Address1, Count1, Source_ptr1, Slave_Address2, Count2, Dest_Ptr2) BYTE;
END;
```

Description:

A data string is sent and received to/from two separate slave devices, separated with a repeat START condition. This has the advantage that the bus does not have to be released with a STOP condition before the transfer from the second slave.

I²C Protocol:

```

L      =Count1
M      =Count2
SlvW1  =Slave_Address1
SlvW2  =Slave_Address2
D1[0..L-1] BASED by Source_Ptr1
D2[0..M-1] BASED by Dest_Ptr2
```

S-SlvW-A-D1[0]-A-D1[1]-.....-A-D1[L-1]-A-S-SlvR-A-D2[0]-A-D2[1]-.....-A-D2[M-1]-N-P

Example:

```

DECLARE Data_Buffer_1(10) BYTE;
DECLARE Data_Buffer_2(4) BYTE;
.....
CALL IIC_Write_Rep_Read (48h, LENGTH(Data_Buffer_1), .Data_Buffer_1, 57h,
  LENGTH(Data_Buffer_2), .Data_Buffer_2);
```

I²C routines for 8XC528

AN438

3.2.13 IIC_Read

Declaration:

IIC_Read:

```
PROCEDURE (Slave_Address, Count, Dest_Ptr) [BIT|BYTE] EXTERNAL;
DECLARE (Slave_Address, Count, Dest_Ptr) BYTE;
END;
```

Description:

IIC_Read is the most basic procedure to read a message from a slave device.

I²C Protocol:

```
M          =Count
D2[0..M-1] BASED by Dest_Ptr
```

```
S-SlvR-A-D2[0]-A-D2[1]-A.....-A-D2[M-1]-N-P
```

Example:

```
DECLARE Data_Buffer(4) BYTE;
.....
CALL IIC_Read (0B5, LENGTH(Data_Buffer), .Data_Buffer);
```

I²C routines for 8XC528

AN438

3.2.14 IIC_Read_Status

Declaration:

IIC_Read_Status:

```
PROCEDURE (Slave_Address, Dest_Ptr) [BIT|BYTE] EXTERNAL;
DECLARE (Slave_Address, Dest_Ptr) BYTE;
END;
```

Description:

Several I²C devices can send a one byte status-word via the bus. IIC_Read_Status can be used for this purpose. IIC_Read_Status works the same way as IIC_Read but the user does not have to pass a count parameter.

I²C Protocol:

Status BASED by Dest_Ptr

S-SlvR-A-Status-N-P

Example:

```
DECLARE Status_Byte BYTE;
.....
CALL IIC_Read_Status (84h, .Status_Byte);
```


I²C routines for 8XC528

AN438

3.2.15 IIC_Read_Sub

Declaration:

IIC_Read_Sub:

```

PROCEDURE (Slave_Address, Count, Dest_Ptr, Sub_Address) [BIT|BYTE] EXTERNAL;
DECLARE (Slave_Address, Count, Dest_Ptr, Sub_Address) BYTE;
END;
```

Description:

IIC_Read_Sub reads a message from a slave device, preceded by a write of the sub-address. Between writing the sub-address and reading the message an I²C restart condition is generated without releasing the bus. This prevents other masters from accessing the slave device in between and overwriting the sub-address.

I²C Protocol:

```

M          =Count
Sub        =Sub_Address
D2[0..M-1] BASED by Dest_Ptr
```

```

S-SlvW-A-Sub-A-S-SlvR-D2[0]-A-D2[1]-A....A-D2[M-1]-N-P
```

Example:

```

DECLARE Data_Buffer(5) BYTE;
.....
CALL IIC_Read_Sub (0A3h, LENGTH(Data_Buffer), .Data_Buffer, 2);
```

I²C routines for 8XC528

AN438

3.2.16 IIC_Read_Rep_Read

Declaration:

IIC_Read_Rep_Read:

```
PROCEDURE (Slave_Address1, Count1, Dest_Ptr1, Slave_Address2, Count2, Dest_ptr2)
  [BIT|BYTE] EXTERNAL;
DECLARE (Slave_Address1, Count1, Dest_ptr1, Slave_Address2, Count2, Dest_Ptr2) BYTE;
END;
```

Description:

Two data strings are read from separate slave devices, separated with a repeat START condition. This has the advantage that the bus does not have to be released with a STOP condition before the transfer from the second slave.

I²C Protocol:

```
L      =Count1
M      =Count2
SlvW1  =Slave_Address1
SlvW2  =Slave_Address2
D1[0..L-1] BASED by Dest_Ptr1
D2[0..M-1] BASED by Dest_Ptr2
```

S-SlvR-A-D1[0]-A-D1[1]-.....-A-D1[L-1]-N-S-SlvR-A-D2[0]-A-D2[1]-.....-A-D2[M-1]-N-P

Example:

```
DECLARE Data_Buffer_1(10) BYTE;
DECLARE Data_Buffer_2(4) BYTE;
.....
CALL IIC_Read_Rep_Read (49h, LENGTH(Data_Buffer_1), .Data_Buffer_1, 51h,
LENGTH(Data_Buffer_2), .Data_Buffer_2);
```

I²C routines for 8XC528

AN438

3.2.17 IIC_Read_Rep_Write

Declaration:IIC_Read_Rep_Write:

```
PROCEDURE (Slave_Address1, Count1, Dest_Ptr1, Slave_Address2, Count2, Source_ptr2)
```

```
[BIT|BYTE] EXTERNAL;
```

```
DECLARE (Slave_Address1, Count1, Dest_ptr1, Slave_Address2, Count2, Source_Ptr2) BYTE;
```

```
END;
```

Description:

A data string is received and send from/to two separate slave devices, separated with a repeat START condition. This has the advantage that the bus does not have to be released with a STOP condition before the transfer from the second slave.

I²C Protocol:

```
L      =Count1
```

```
M      =Count2
```

```
SlvW1   =Slave_Address1
```

```
SlvW2   =Slave_Address2
```

```
D1[0..L-1] BASED by Dest_Ptr1
```

```
D2[0..M-1] BASED by Source_Ptr2
```

```
S-SlvR-A-D1[0]-A-D1[1]-.....-A-D1[L-1]-N-S-SlvW-A-D2[0]-A-D2[1]-.....-A-D2[M-1]-A-P
```

Example:

```
DECLARE Data_Buffer_1(10) BYTE;
```

```
DECLARE Data_Buffer_2(4) BYTE;
```

```
.....
```

```
CALL IIC_Read_Rep_Write (49h, LENGTH(Data_Buffer_1), .Data_Buffer_1, 58h,  
LENGTH(Data_Buffer_2), .Data_Buffer_2);
```

I²C routines for 8XC528

AN438

3.2.18 Slave mode routines

There are two ways for the I²C interface to enter the slave-mode:

- After an I²C interrupt the software must enter the slave-receiver mode to receive the slave address. This address will then be compared with its own address. If there is a match either slave-transmitter or slave-receiver mode will be entered. If no match occurs, the interrupted program will be continued.
- During transmission of a slave-address in master-mode, arbitration is lost to another master. The interface must then switch to slave-receiver mode to check if this other master wants to address the 8xC528 I²C interface.

The slave-mode protocol is very application dependent. In this note the basic slave-receive and slave-transmit routines are given and should be considered as examples. The user may for instance send NO_ACK after receiving a number of bytes to signal to the master-transmitter that a data buffer is full. A description of the code will be given later.

Slave parameters are given with the Init_IIC procedure. The passed parameters are the own-slave-address and a source/destination-pointer to a data buffer.

The slave-routine will be suspended at the following conditions:

- Interrupts with higher priority. Slave-routine will be resumed again after interrupt is handled.
- If a NO_ACKNOWLEDGE is received from a master-receiver.
- If a STOP condition is detected from a master transmitter.

Constraints for user software:

- The user must control the global enable (EA) bit.
- The user must control the priority level of the I²C interrupt. If the slave routine is interrupted by a higher priority interrupt, the SCL line will be stretched to postpone bus transfer until the higher interrupt is finished.

I²C routines for 8XC528

AN438

3.3 THE SLAVE ROUTINE: SLAVE.ASM

On page 30 the listing of the slave routine can be seen. The routine is written in such a way that stretching of SCL is minimized. Application code can be inserted in this routine and this will increase stretching time.

The routine has 2 entry points.

Entry via `MST_ENTRY` happens when an arbitration error has occurred when transmitting a slave address in master mode. Auto-clock generation will be disabled and SCL stretching enabled. The byte will be continued to be received and can later be compared with the own slave address.

The second entry point is via an interrupt when a START condition is detected. At `PIP0A` the context of the interrupted program is stored. Next Auto-clock generation is disabled and SCL stretching enabled. Reception of the slave address can now begin by calling `RCV_SL_BY`. When the received slave-address is compared with the own-slave-address the R/W-bit is ignored. If there is no match between the 2 addresses, a negative ACK bit is sent and the slave routine is left via `EXIT`. If there was a match the R/W bit is checked to enter the slave-receiver or slave-transmitter mode.

The slave-transmitter mode starts at `NXT_TRX`. After getting the byte from the data buffer via `BUF_POINT` and initialising the bit counter `BIT_CNT` the transmission loop is entered. A bit is written via access to `S1BIT` because this will automatically reset the `CLH` and `WBF` status flags, and also SCL stretching. Now `WBF` must be tested until the transmission is successful. When `WBF` becomes true SCL will be stretched again. When 8 bits are sent the SDA line is released and `RBF` is tested until the ACK bit is received. The ACK bit is read by reading `SDI` instead of `S1BIT` to maintain SCL stretching. If ACK was false no more bytes have to be sent and the routine is left. If another byte has to be transmitted, `BUF_POINT` is updated and transmission will continue.

The slave-receiver mode starts at `RCV_SLAVE`. A byte is received by calling `RCV_SL_BY`. This routine will clear the `CY`-flag when a STOP condition has been received. This means that the master will send no more bytes to this slave and the slave routine will be left. When no STOP condition was detected the received byte will be stored @`BUF_POINT` and an ACK bit will be sent. After this a new byte can be received.

When calling `RCV_SL_BY` the bit counter `BIT_CNT` will be initialized and the SCL stretching stopped by a dummy access to `S1BIT`. In the receive loop both `BB` and `RBF` will be checked. When `BB` is cleared, a STOP condition is detected and the routine will be left with `CY=0`. The first 7 bits are received via `S1BIT` because this will release stretching. The 8th bit is accessed via `SDI` because stretching must be maintained.

If the slave routine is left via `EXIT`, the `STR` bit is cleared (to disable stretching on SCL edges when the 8xC528 is not addressed as slave) and a dummy access to `S1BIT` is done to finish current SCL stretching. If the slave routine was entered via an interrupt the previous context is restored.

I²C routines for 8XC528

AN438

TSW-ASM51 V3.0b Serial #00052252 Slave interrupt routine
PAGE 1

LOC	OBJ	LINE	SOURCE
		1	\$TITLE(Slave interrupt routine)
		2	\$DEBUG
		3	\$NOLIST
		6	;
		7	;This routine handles I2C interrupts.
		8	;8xC528 I2C interface enters in slave mode.
		9	;After testing R/W bit, 8xC528 will go in slave-transmit or
		10	;slave-receive mode.
		11	;Source or destination buffer for data uses pointer SLAVE_SUB_ADDRESS
		12	;Slave routine will use register bank 01
		13	;
		14	;*****
		15	;Interrupt entry point
		16	
		17	CSEG AT 53H
		18	
0053: 020000	R	19	LJMP __PIPOA ;Vector to interrupt handler
		20	;
		21	;*****
		22	
		23	I2C_DRIVER SEGMENT CODE INBLOCK
		24	RSEG I2C_DRIVER
		25	
		26	PUBLIC MST_ENTRY
		27	EXTRN DATA(SLAVE_SUB_ADDRESS)
		28	EXTRN BIT(ARB_LOST)
		29	
REG END		30	BUF_POINT SET R0
REG END		31	OWN_SLAVE SET R7
REG END		32	BIT_CNT SET R2
		33	
		34	;*****
		35	
0000: C0E0	R	36	__PIPOA:PUSH ACC ;Push CPU status on stack
0002: C0D0		37	PUSH PSW
0004: 75D008		38	MOV PSW,#08H ;Select registerbank 01
		39	
		40	;*****
		41	;Check slave address
		42	;*****
		43	
0007: 43D842		44	ORL S1SCS,#01000010B ;Disable SCL generation and enable SCL
			;stretching stretching
000A: 1142	R	45	ACALL RCV_SL_BY ;Receive slave address, on exit SCL is
			; stretched
000C: A2E0		46	PROC: MOV C,ACC.0 ;Store R/W bit in F0
000E: 92D5		47	MOV F0,C
0010: 6F		48	XRL A,OWN_SLAVE ;Compare received slave address

I²C routines for 8XC528

AN438

TSW-ASM51 V3.0b Serial #00052252 Slave interrupt routine

PAGE 2

LOC	OBJ	LINE	SOURCE
0011:	C2E0	49	CLR ACC.0 ;Ignore R/W bit
0013:	7050	50	JNZ NO_MATCH ;Leave slave-routine if there is no match
0015:	C3	51	CLR C ;Send ACK
0016:	115C	R 52	ACALL SEND_ACK
0018:	A800	R 53	MOV BUF_POINT,SLAVE_SUB_ADDRESS ;Get buffer pointer
001A:	A2D5	54	MOV C,F0 ;Restore R/W bit
001C:	5019	55	JNC RCV_SLAVE ;Test R/W bit
		56	
		57	
		58	*****
		59	;Slave transmitter mode
		60	*****
		61	
		62	
001E:	E6	63	NXT_TRX:MOV A,@BUF_POINT;Get byte to send
001F:	7A08	64	MOV BIT_CNT,#08 ;Init bit counter
		65	
0021:		66	NXT_TRX_BIT:
0021:	F5D9	67	MOV S1BIT,A ;Trx bit and stretch after transmission
0023:	23	68	RL A ;Prepare next bit to send
0024:	30DAFD	69	JNB WBF,\$;Test if bit is sent
0027:	DAF8	70	DJNZ BIT_CNT,NXT_TRX_BIT ;Test if all bits are sent
		71	
0029:	D2DF	72	SETB SDO ;Release SDA line for NO_ACK/ACK reception
002B:	E5D9	73	MOV A,S1BIT ;Stop stretching
002D:	30DBFD	74	JNB RBF,\$;Test is ACK bit is received
0030:	A2DF	75	MOV C,SDI ;Read bit, SCL remains stretched
0032:	4040	76	JC EXIT ;NO_ACK received. Exit slave routine
0034:	08	77	INC BUF_POINT ;ACK received. Update pointer for next byte to
			;send
0035:	80E7	78	SJMP NXT_TRX
		79	
		80	*****
		81	;Slave receiver mode
		82	*****
		83	
0037:		84	RCV_SLAVE: ;Entry in slave-receiver mode
0037:	1142	R 85	ACALL RCV_SL_BY ;Receive byte
0039:	5039	86	JNC EXIT ;If STOP is detected, then exit
003B:	F6	87	MOV @BUF_POINT,A;Store received byte
003C:	C3	88	CLR C ;Send ACK
003D:	115C	R 89	CALL SEND_ACK
003F:	08	90	INC BUF_POINT ;Update pointer
0040:	80F5	91	SJMP RCV_SLAVE ;Receive next byte
		92	
		93	

I²C routines for 8XC528

AN438

TSW-ASM51 V3.0b Serial #00052252 Slave interrupt routine

PAGE 3

```

LOC   OBJ          LINE   SOURCE
;*****
94 ;*****
95 ;Receive byte routine
96 ;On exit, received byte in accu
97 ;On exit CY=0 if STOP is detected
98 ;*****
0042: 99 RCV_SL_BY:
0042: 7A08          100     MOV BIT_CNT,#08
0044: E5D9          101     MOV A,S1BIT      ;Disable stretching from START or previous ACK
0046: E4             102     CLR A
0047:             103 RCV_BIT:
0047: 30DC10         104     JNB BB,STOP_RCV ;Test if STOP-condition is received
004A: 30DBFA         105     JNB RBF,RCV_BIT ;Wait till received bit is valid
004D: BA0105         106     CJNE BIT_CNT,#01,ASSEM_BIT ;Check if last bit is to be received
                                107
0050: A2DF           108     MOV C,SDI        ;Get last bit without stopping stretching
0052: 33             109     RLC A
0053: D3             110     SETB C           ;No STOP detected
0054: 22             111     RET
                                112
0055:             113 ASSEM_BIT:
0055: 45D9           114     ORL A,S1BIT      ;Receive bit; release RBF,CLH and SCL stretching
                                115
0057: 23             115     RL A
0058: DAED           116     DJNZ BIT_CNT,RCV_BIT
                                117
005A:             118 STOP_RCV:
005A: C3             119     CLR C           ;STOP detected
005B: 22             120     RET
                                121
122 ;*****
123 ;Send ACK/NO_ACK. Value of ACK in Carry
124 ;*****
005C:             125 SEND_ACK:
005C: 13             126     RRC A
005D: F5D9           127     MOV S1BIT,A      ;Carry to SDA line
005F: 30DAFD         128     JNB WBF,$         ;Test if ACK/NO_ACK is sent
0062: D2DF           129     SETB SDO         ;Release SDA line
0064: 22             130     RET
                                131
132 ;*****
133 ;No match between received slave-address and own-slave-address
134 ;*****
0065:             135 NO_MATCH:
0065: D3             136     SETB C           ;Send NO_ACK
0066: 115C           R 137     ACALL SEND_ACK
0068: 800A           138     SJMP EXIT

```

I²C routines for 8XC528

AN438

TSW-ASM51 V3.0b Serial #00052252 Slave interrupt routine

PAGE 4

```

139 ;*****
141 ;Entry point when an arbitration-lost condition is detected in
;master-mode.
142 ;*****
006A: 143 MST_ENTRY:
006A: 23 144 RL A ;Restore slave address sofar
006B: C2E0 145 CLR ACC.0
006D: 43D842 146 ORL S1SCS,#01000010B ;Disable SCL generation and enable SCL
;stretching
0070: 1147 R 147 ACALL RCV_BIT ;Proceed with receiving rest of slave address
0072: 8098 148 SJMP PROC
149
150
151 ;*****
152 ;Exit from interrupt routine
153 ;*****
154
0074: C2D9 155 EXIT: CLR STR ;Disable stretching on next falling SCL edges
0076: E5D9 156 MOV A,S1BIT ;Stop current SCL stretching
0078: 300001 R 157 JNB ARB_LOST,EX_SL
007B: 22 158 RET ;Exit when entered from master mode
007C: D0D0 159 EX_SL: POP PSW ;Restore old CPU status
007E: D0E0 160 POP ACC
0080: 32 161 RETI
162
0081: 163 END

```

I²C routines for 8XC528

AN438

4. EXAMPLES**4.1 INTRODUCTION**

Some examples are given how to use the I²C routines in an application program. Examples are given for an assembly, PL/M and C program.

The program displays time from the PCF8583P clock/calendar/RAM on an LCD display driven by the PCF8577.

The example can be executed on the OM4151 I²C evaluation board.

4.2 Using the routines with assembly sources

From page 35 the listing is shown of the example program. The most important aspect when using the I²C routines, is preparing the input parameters before the sub-routine call.

When for example the IIC_Write routine must be called, the parameters must be called in the following order:

```
MOV _IIC_READ_BYTE,#SLAVE_ADR
MOV _IIC_READ_BYTE+1,#COUNT_1
MOV _IIC_READ_BYTE+2,#SOURCE_PTR_1
CALL _IIC_READ
```

Note that the order of defining the parameters is the same as in a PL/M-call (see page 22).

An easier way to call the routines is making a macro that includes the initialising of the parameters.

The example program makes use of macros.

IIC_Read is then called in the following way:

```
%IIC_Read(Slave_Adr,Count_1,Source_Ptr_1);
```

Note that in the listing the contents of the macro are shown, in stead of the call.

The macro must be written as follows:

```
%* DEFINE (IIC_Read(SLAVE_ADR,COUNT_1,SOURCE_PTR_1))
MOV _IIC_READ_BYTE,#%SLAVE_ADR
MOV _IIC_READ_BYTE+1,#%COUNT_1
MOV _IIC_READ_BYTE+2,#%SOURCE_PTR_1
LCALL _IIC_READ
```

Macro's for the I²C CALL's are found in I2C.MAC. This file should be included in all modules making use of the macro's. One of the modules should also include the variable definitions needed by the I²C routines. These are found in file VAR_DEF.ASM. If the program consists of more than 1 module, then these modules should also include EXT_VAR.ASM. This file contains the EXTRN-definitions of the I²C routines.

When an I²C routine is called the accumulator contains status information and the CY-bit is set if an error has occurred. The contents of the accumulator are the same as the returned byte when using PL/M (see pg. 10).

I²C routines for 8XC528

AN438

TSW-ASM51 V3.0b Serial #00052252 Assembly example program

PAGE 1

LOC	OBJ	LINE	SOURCE
		1	\$TITLE(Assembly example program)
		2	\$DEBUG
		3	
		4	;Hours and minutes will be displayed on LCD display
		5	;Dot between hours and minutes will blink
		6	
		7	\$;Include I2C var. definitions
		8	# 1 "C:\USER\VAR_DEF.ASM"
		74	# 8 "DEMO_ASM.ASM"
		8	\$;Include I2C macro's
		9	# 1 "C:\USER\I2C.MAC"
		35	# 9 "DEMO_ASM.ASM"
00A2		10	CLOCK_ADR EQU 0A2h ;Address of PCF8583
0001		11	CL_SUB_ADR EQU 01h ;Sub address for reading time
0074		12	LCD_ADR EQU 74h ;Address of PCF8577
		13	
		14	RAMVAR SEGMENT DATA ;Segment for variables
		15	USER SEGMENT CODE ;Segment for application
			;program
		16	
		17	RSEG RAMVAR
0000:	R	18	STACK: DS 10 ;Stack area
000A:		19	TIME_BUFFER:DS 4 ;Buffer for I2C strings
000E:		20	LCD_BUFFER: DS 5
		21	
		22	CSEG AT 00
0000: 020000	R	23	LJMP APL_START
		24	
		25	
		26	RSEG USER
		27	
0000:	R	28	APL_START:
0000: 900073	R	29	MOV DPTR,#LCD_TAB ;Pointer to segment table
0003: 7581FF	R	30	MOV SP,#STACK-1 ;Initialise stack
0006: 750E00	R	31	MOV LCD_BUFFER,#00 ;Control word for LCD driver
		32	
0009: 750022	R	33	MOV _Init_IIC_Byte ,#22h
000C: 75010A	R	34	MOV _Init_IIC_Byte+1,#TIME_BUFFER
000F: 120000	R	35	LCALL _Init_IIC
		36	;Initialise I2C interface
0012: E4		37	CLR A ;Prepare buffer for clock int.
0013: F50A	R	38	MOV TIME_BUFFER,A
0015: F50B	R	39	MOV TIME_BUFFER+1,A
		40	
0017: 7500A2	R	41	MOV _IIC_Write_Byte ,#CLOCK_ADR
001A: 750102	R	42	MOV _IIC_Write_Byte+1,#2
001D: 75020A	R	43	MOV _IIC_Write_Byte+2,#TIME_BUFFER
0020: 120000	R	44	LCALL _IIC_Write

I²C routines for 8XC528

AN438

TSW-ASM51 V3.0b Serial #00052252 Assembly example program PAGE 2

LOC	OBJ	LINE	SOURCE
		45	;Initialise clock
		46	
0023:		47	REPEAT:
0023: 7500A2	R	48	MOV _IIC_Read_Sub_Byte ,#CLOCK_ADR
0026: 750104	R	49	MOV _IIC_Read_Sub_Byte+1,#4
0029: 75020A	R	50	MOV _IIC_Read_Sub_Byte+2,#TIME_BUFFER
002C: 750301	R	51	MOV _IIC_Read_Sub_Byte+3,#CL_SUB_ADR
002F: 120000	R	52	LCALL _IIC_Read_Sub
		53	;Read time
		54	
		55	;Time has been read. Order: hundreds of sec's, sec's, min's and hr's
0032: E50D	R	56	MOV A,TIME_BUFFER+3 ;Mask of hour counter
0034: 543F		57	ANL A,#3Fh
0036: F50D	R	58	MOV TIME_BUFFER+3,A
		59	
0038: 120054	R	60	CALL CONVERT ;Convert time data to LCD ;segment data
		61	
		62	;Check if dot has to be switched on
003B: 431101	R	63	ORL LCD_BUFFER+3,#01h
		64	;If lsb of seconds is '0', then switch on dp
003E: E50B	R	65	MOV A,TIME_BUFFER+1 ;Get seconds
0040: 13		66	RRC A
0041: 4003		67	JC PROCEED
0043: 430F01	R	68	ORL LCD_BUFFER+1,#01 ;Switch on dp
		69	
		70	;Display new time
0046:		71	PROCEED:
0046: 750074	R	72	MOV _IIC_Write_Byte ,#LCD_ADR
0049: 750105	R	73	MOV _IIC_Write_Byte+1,#5
004C: 75020E	R	74	MOV _IIC_Write_Byte+2,#LCD_BUFFER
004F: 120000	R	75	LCALL _IIC_Write
		76	
0052: 80CF		77	SJMP REPEAT ;Read new time
		78	
		79	
		80	;CONVERT converts BCD data of time to segment data
0054: 780F	R	81	CONVERT:MOV R0,#LCD_BUFFER+1 ;R0 is pointer
0056: E50D	R	82	MOV A,TIME_BUFFER+3 ;Get hours
0058: C4		83	SWAP A ;Swap nibbles
0059: 12006D	R	84	CALL LCD_DATA ;Convert 10's of hours
005C: E50D	R	85	MOV A,TIME_BUFFER+3
005E: 12006D	R	86	CALL LCD_DATA ;Convert hours
0061: E50C	R	87	MOV A,TIME_BUFFER+2 ;Get minutes
0063: C4		88	SWAP A
0064: 12006D	R	89	CALL LCD_DATA ;Convert 10's of minutes
0067: E50C	R	90	MOV A,TIME_BUFFER+2
0069: 12006D	R	91	CALL LCD_DATA ;Convert minutes

I²C routines for 8XC528

AN438

TSW-ASM51 V3.0b Serial #00052252 Assembly example program

PAGE 3

```

LCC  OBJ      LINE  SOURCE
006C: 22      92      RET
                        93      ;Initialise clock
006D:          94      ;LCD_DATA gets data from segment table and stores it in LCD_BUFFER
006D:          95      LCD_DATA:
006D: 540F      96      ANL A,#0FH      ;Mask off LS-nibble
006F: 93      97      MOVC A,@A+DPTR      ;Get segment data
0070: F6      98      MOV @R0,A      ;Save segment data
0071: 08      99      INC R0
0072: 22     100      RET
                        101      ;
0073:          102      ;Conversion table for LCD
0073:          103      LCD_TAB:
0073: FC60DA    104      DB 0FCH,60H,0DAH      ;'0','1','2'
0076: F266B6    105      DB 0F2H,66H,0B6H      ;'3','4','5'
0079: 3EE0FE    106      DB 3EH,0E0H,0FEH      ;'6','7','8'
007C: E6      107      DB 0E6H      ;'9'
                        108      ;
007D:          109      END

```

I²C routines for 8XC528

AN438

4.3 Using the routines with PL/M-51 sources

The listing from pg. 39 shows the listing of the clock program in PL/M-51. The procedures are untyped. The routines are used the same way as in the examples of chapter 3.2

```

Clock: Do;
/* Variable and constant declarations */
Declare Time_Buffer(4) Byte Main;
Declare LCD_Buffer(5) Byte Main;
Declare Tab_Point_Word Main;
Declare (LCD_Point,Time_Point) Byte Main;
Declare Segment_Based_LCD_Point Byte Main;
Declare Time_Based_Time_Point Byte Main;
Declare Tab_Value_Based_Tab_Point Byte Constant;

$OPTIMIZE(4)
$DEBUG
$CODE

/* Hours and minutes will be displayed on LCD display
   Dots between hours and minutes will blink */

Demo_plm: Do;

/* External declarations */
Init_IIC: Procedure(Own_Adr,Slave_Ptr) External;
        Declare (Own_Adr,Slave_Ptr) Byte Main;
End Init_IIC;

IIC_Write: Procedure(Sl_Adr,Nr_Bytes,Source_Ptr) External;
        Declare (Sl_Adr,Nr_Bytes,Source_Ptr) Byte Main;
End IIC_Write;

IIC_Read_Sub: Procedure(Sl_Adr,Nr_Bytes,Dest_Ptr,Sub_Adr) External;
        Declare(Sl_Adr,Nr_Bytes,Dest_Ptr,Sub_Adr) Byte Main;
End IIC_Read_Sub;

        Call IIC_Write(Clock_Adr,2,Time_Buffer); /* Initialize clock */
        Do While LCD_Buffer(0)=0; /* LCD control word */
        Time_Buffer(0)=0;
        LCD_Buffer(0)=0;
        Call IIC_Write(Clock_Adr,2,Time_Buffer); /* Initialize clock */
        Do While LCD_Buffer(0)=0; /* Program loop */
        Call IIC_Read_Sub(Clock_Adr,4,Time_Buffer,Clock_Adr);
        /* Get time */
        LCD_Point=LCD_Buffer(1); /* Init LCD_Buffer(1) */
        Time_Point=Time_Buffer(2);
        Tab_Point=LCD_Tab(0)+SER(Time,1); /* 10-MS */
        Segment=Tab_Value;
        LCD_Point=LCD_Point+1;
        Tab_Point=LCD_Tab(0)+(Time AND 0FH); /* MS */
        Segment=Tab_Value;
        Time_Point=Time_Point+1;
        LCD_Point=LCD_Point+1;
        Tab_Point=LCD_Tab(0)+SER(Time,1); /* 10-MS */
        Segment=(Tab_Value OR 0FH); /* 4p */
        LCD_Point=LCD_Point+1;
        Tab_Point=LCD_Tab(0)+(Time AND 0FH); /* MS */
        Segment=Tab_Value;
        Time_Point=Time_Buffer(1)+1; /* Check sec's for blinking */
        LCD_Point=LCD_Buffer(1);
        If (Time AND 01H)=0 then Segment=(Segment OR 01H);
        Call IIC_Write(LCD_Adr,5,LCD_Buffer); /* Display time */
        End;
End Demo_plm;
End Clock;

```

I²C routines for 8XC528

AN438

```

Clock: Do;
    /* Variable and constant declarations */
    Declare LCD_TAB(*) Byte Constant (0FCh,60H,0DAH,0F2H,66H,
                                     0B6H,3EH,0EOH,0FEH,0E6H);

    Declare Time_Buffer(4) Byte Main;
    Declare LCD_Buffer(5) Byte Main;
    Declare Tab_Point Word Main;
    Declare (LCD_Point,Time_Point) Byte Main;
    Declare Segment Based LCD_Point Byte Main;
    Declare Time Based Time_Point Byte Main;
    Declare Tab_Value Based Tab_Point Byte Constant;

    Declare Clock_Adr Literally '0A2h';
    Declare LCD_Adr Literally '74h';
    Declare Cl_Sub_Adr Literally '01h';

    Call Init_IIC(22h,.Time_Buffer);
    LCD_Buffer(0)=0; /* LCD control word */
    Time_Buffer(0)=0;
    Time_Buffer(1)=0;
    Call IIC_Write(Clock_Adr,2,.Time_Buffer); /* Initialise clock */

    Do While LCD_Buffer(0)=0; /* Program loop */
        Call IIC_Read_Sub(Clock_Adr,4,.Time_Buffer,Cl_Sub_Adr);
        /* Get time */
        LCD_Point=.LCD_Buffer+1; /* Initialise pointers */
        Time_Point=.Time_Buffer(3);
        Tab_Point=.LCD_Tab(0)+SHR(Time,4); /* 10-HR's */
        Segment=Tab_Value;

        LCD_Point=LCD_Point+1;
        Tab_Point=.LCD_Tab(0)+(Time AND 0FH); /* HR's */
        Segment=Tab_Value;
        Time_Point=Time_Point-1;
        LCD_Point=LCD_Point+1;
        Tab_Point=.LCD_Tab+SHR(Time,4); /* 10-MIN's */
        Segment=(Tab_Value OR 01H); /* dp */
        LCD_Point=LCD_Point+1;
        Tab_Point=.LCD_Tab+(Time AND 0FH); /* MIN's */
        Segment=Tab_Value;
        Time_Point=.Time_Buffer(1)+1; /* Check sec's for blinking */
        LCD_Point=.LCD_Buffer+1;
        If (Time AND 01H)>0 then Segment=(Segment OR 01H);
        Call IIC_Write(LCD_Adr,5,.LCD_Buffer); /* Display time */
    End;

End Clock;

End Demo_plm;

```


I²C routines for 8XC528

AN438

4.4 Using the routines with C sources

On page 42 an example of a C program is shown using the I²C routines. Function prototypes are found in header file "i2c.h". In this example the function prototypes are written in such a way that no value is returned by the function. If the STATUS byte is needed, the header file may be changed to return a byte.

Note that the function calls are written in upper-case. This is due to the fact that the used version of the assembler/linker is case sensitive.

```
#include <C:\USER\i2c.h>

rom char LCD_Tab[]={0xFC,0x60,0xDA,0xF2,0x66,0xB6,0x3E,0xE0,0xFE,0xE6};

void main()

#define Clock_Adr      0xA2
#define LCD_Adr        0x74
#define Cl_Sub_Adr     0x01

rom char * Tab_Ptr;
data char Time_Buffer[4];
data char * Time_Ptr;
data char LCD_Buffer[5];
data char * LCD_Ptr;

INIT_IIC(0x22,&Time_Buffer);
LCD_Buffer[0]=0; /* LCD control word */
Time_Buffer[0]=0;
Time_Buffer[1]=0;
IIC_WRITE(Clock_Adr,2,&Time_Buffer); /* Initialise clock */

while (1) /* Program loop */
{
    IIC_READ_SUB(Clock_Adr,4,&Time_Buffer,Cl_Sub_Adr);
    /* Get time */

    LCD_Ptr = &LCD_Buffer[1]; /* Initialise pointers */
    Time_Ptr = &Time_Buffer[3];
    Tab_Ptr = (LCD_Tab+(*Time_Ptr >> 4)); /* 10-HR's */
    *(LCD_Ptr++) = *Tab_Ptr;
    Tab_Ptr = (LCD_Tab+(*Time_Ptr-- & 0x0F)); /* HR's */
    *(LCD_Ptr++) = *Tab_Ptr;
    Tab_Ptr = (LCD_Tab+(*Time_Ptr >> 4)); /* 10-MIN's */
    *(LCD_Ptr++) = (*Tab_Ptr | 0x01); /* dp */
    Tab_Ptr = (LCD_Tab+(*Time_Ptr & 0x0F)); /* MIN's */
    *LCD_Ptr = *Tab_Ptr;
    Time_Ptr = &Time_Buffer[1]; /* Check sec's for blinking */
    LCD_Ptr = &LCD_Buffer[1];
    if ((*Time_Ptr & 0x01)>0)
        *LCD_Ptr = (*LCD_Ptr | 0x01);
    IIC_WRITE(LCD_Adr,5,&LCD_Buffer); /* Display time */
}
```

I²C routines for 8XC528

AN438

5: CONTENTS OF DISK

A disk contains the following 3 directories:

1: \USER

This directory contains the files that may be used in the user program.

I2C_DR.LIB Library with I²C routines.

I2C.H Header file for C applications.

I2C.MAC Macro's for the I²C routine calls in assembly programs.

VAR_DEF.ASM Include file with variable definitions for assembly programs.

EXT_VAR.ASM Include file with external definitions for assembly programs.

LIB.BAT Example batch file to create I2C_DR.LIB.

ASM.BAT Example batch file to assemble source modules for library.

2: \EXAMPLE

This directory contains the source files of the examples described in chapter 4.

DEMO_ASM.* Assembly example.

DEMO_PLM.* PL/M example.

HEAD_51.SRC Example of environment file for PL/M example.

DEMO_C.* C example.

CSTART.ASM Example of environment file for C example.

3: \SOURCE

This directory contains the source files of the modules in the library.

Using the P82B715 I²C extender on long cables AN444

Author: Don Sherman, Sunnyvale

The P82B715 I²C Buffer was designed to extend the range of the local I²C bus out to 50 Meters. This application note describes the results of testing the buffer on several different types of cables to determine the maximum operating distances possible. The results are summarized in a table for easy reference.

The I²C bus was originally conceived as a convenient 2 wire communication method between Integrated Circuits located within a common chassis, such as inside a TV set or inside a VCR. The serial protocol contains an address, or identifying code, for each type of device and additional internal addresses, if needed within the addressed device. Each device has its own decoding circuitry to allow it to recognize its own unique address or identifying code. To communicate, a device watches the bus activity and jumps in when it sees a stop. Once a Master gets control of the bus, it sends the address of the particular device with which it wants to communicate. Communication will then transpire between the Master and the Slave device. The existence of many types of ICs which have built-in I²C interface capabilities makes system design almost as easy as drawing a block diagram. Real-time clocks, RAM, A/D converters, EEPROMs, Microcontrollers, Keyboard encoders, LCD display drivers, and many other I²C supported chips all communicate over two wires rather than needing 16 Address lines, 8 data lines and Address decoders along with handshake signals, which more conventional designs would require to be routed all over the Printed Circuit board.

Now, with the introduction of the I²C buffer chip, it is easy to branch out beyond the single chassis mode and use this convenient local area network to tie together whole systems without the need to convert from the "internal" I²C protocol to an external communication medium such as RS-232 and then RS-485. By using the new Philips I²C buffer, the external systems' components can be accessed as easily as the internal I²C connected components.

The P82B715 is an 8 pin IC which contains 2 identical amplifier sections to allow for the current amplification and buffering of both the SDA and the SCL signals on the I²C bus. Each section in the P82B715 contains a bipolar times 10 current amplifier which senses the direction of current flow through an internal 30 ohm series resistor in the I²C line. The P82B715 then boosts the current, while keeping the voltage gain at unity, and continues to maintain the voltage drop direction across the resistor. This

configuration results in different waveforms as the P82B715 starts to do its job. If the driving source has a strong current sink capability, then it will start to drive the buffered I²C line immediately through the 30 ohm resistor. A microsecond later the P82B715's amplified pull down current kicks in and pulls the line down even harder. If the driving IC is only capable of the I²C specified 3 milliamp pull down current, the buffered bus will fall a little and then just wait at that voltage level for the propagation delay of the amplifier to finally turn on and bring the buffered bus down to a logic low. Thus, there will always be some form of a step in the falling edge of the buffered output waveform, see Figure 1. A weak source will have a step (plateau) up near 4 volts and a strong source, such as the Philips Semiconductors 87C751 microcontroller, will have the step occur below 2 volts. The position of the step will be determined by the current sink capability of the I²C bus driver versus the value of the pull-up resistor which is used on the buffered I²C bus, $V_{step} = 5V - (I_{sink} \times R_{buf})$. For example: $V_{step} = 5V - (3mA \times .165k\text{ ohms}) = 5 - .495 = 4.5\text{Volts}$; another example: $V_{step} = 5V - (20mA \times .165k\text{ ohms}) = 5 - 3.3 = 1.7\text{Volts}$.

Running the I²C signals over long distances poses several problems. The I²C SDA and SCL lines are monitored by all of the ICs connected on the I²C bus. These ICs each have their own circuitry to decipher the information on the bus. In normal operation, a Start occurs when there is a high to low transition on the SDA line while SCL is high. Obviously, if any external noise is coupled into the SDA line, it could be mistakenly perceived as a Start. Because of this, some form of shielding will be preferred to protect the two I²C signals from external noise sources. During the transmission of data there are signals which are active on both SDA and SCL. If these normal signals are cross-coupled, then data can be corrupted. Thus, although the standard telephone twisted pair cable is the most commonly available built in cable, it is not recommended for long I²C runs. This cable maximizes crosstalk, due to the twisted pair configuration and, since there is no shielding, is very vulnerable to adjacent wire telephone signal coupling and to any stray external electromagnetic interference. This effect can be somewhat reduced by running a signal wire and a grounded wire as adjacent pairs.

Long distance cables present capacitive loading which must be overcome with the driver chips. The limiting factor is the amount of pull-up current which is available to charge the line capacitance. With the simple resistor

pull-up recommended by I²C standards, three milliamps is available for charging this line capacitance. The rise time of the signal will increase linearly with the increase in capacitive loading and the specified maximum capacitive loading is only 400 Pico Farads for guaranteed 100kHz communication rates. The P82B715 current buffer allows for 30 milliamps of pull-up current, with a resulting maximum capacitive loading of 4,000 Pico Farads (4 Nano Farads).

The I²C hardware inputs look at the I²C signals and act when those signals pass through the active linear region at about 1.2 to 1.4 volts, and are detected as digital levels. Thus, there is a delay between when an output transistor turns off and when the rising signal is detected as a logic one at the receiver. This time depends on the value of the pull-up resistor, the perceived capacitance at the transmitting end, the delay through the cable, and finally the delay through the receiver's amplifier to its output stage. The maximum allowable time is limited by the characteristic that the I²C master provides the clock signal which must travel down the cable and be received by the slave. This slave must act on the clock signal and produce data information which is sent back to the master with an additional set of delays. Upon reception the data must be put in its proper place before the master starts its next clock signal, or an error will occur.

Different types of cable were tested and the results are shown in Table 1. Keep in mind that the results are based on cable runs in a low electrical noise environment. If reliable operation is desired in a high electrical noise environment, shielded cable must be used. For "short" runs, flat cable with every other conductor grounded, seems to provide a good, low capacitance medium for I²C transmission, otherwise, the shielded audio cable seemed to provide the best price/performance. Note that for long runs, it is desirable to have a separate power supply at each end of the cable, and the shield or ground wire will provide a common reference between the two supplies. The voltage drop due to the resistance of the wire usually is the limiting factor for very long runs of cable where the power to the remote system must also come through the cable. Table 1 shows the results of testing with longer and longer cable lengths until failures were detected. The values in the table represent the maximum cable lengths which still provided error free code from a modified version of the Ping-pong program which is listed in Application Note AN430.

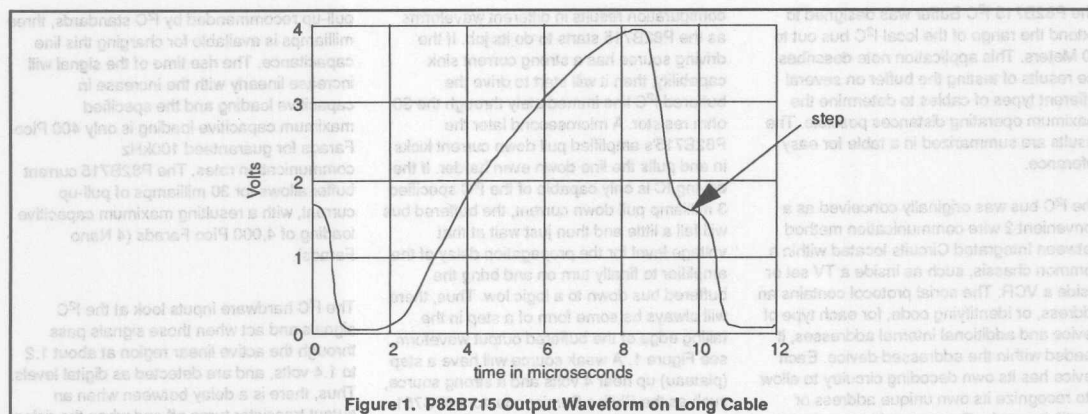
Using the P82B715 I²C extender on long cables AN444

Figure 1. P82B715 Output Waveform on Long Cable

Table 1. Test Results with P82B715 Over Long Cables

CABLE TYPE	Ohms/m	pF/m	Total Length	Total Ohms	Total Cap.
Belden 8723 45 Ohm Audio 2 each 2—24AWG wire stranded Beldfoil Aluminum- polyester shielded with common drain wire SDA & ground on one pair; SCL & ground on other pair	.049	115	305M (1000')	11.5	48.2nF
Belden 8723 45 Ohm Audio using 1 shielded pair, SDA on Red, SCL on Black	.049	115	330M (1100')	12.7	53nF
RG-174/U 50 Ohm Video Cable SDA and grounded shield in one cable SCL and grounded shield in one cable	.318	101	150M (500')	47.7	15.2nF
"Telephone Cable" 22&24 AWG Solid Copper Twisted Pair, Level 3 LAN & Medium Speed Data SDA and ground in one twisted pair SCL and ground in one twisted pair	.0286	66	95M (310')	2.7	6.4nF
Flat "Ribbon" Cable, every other conductor grounded	.20	52	400M (1320')	80.5	21nF

In all of the tests, the power supply voltage was 4.5 volts. The ground for the remote test fixture was through the long cable. Since 4.5 volts is the recommended minimum voltage for both the 87C751 and the P82B715, it was not possible to operate the remote unit on power supplied through the long cable, since any ohmic drop would place the ICs out of their specified range. However, it is necessary to connect the grounds between the two units for the best noise immunity.

The P82B715 is designed to drive a 4 nF capacitive load at 100kHz. However, the actual total capacitances of the long cables which worked were substantially greater than this. The loading did effect the software driven hardware part of the 87C751. To achieve a true 100kHz data rate, it was necessary to shorten the 751 Timer values for the I²C drivers. This resulted in an asymmetrical waveform, but did achieve a 10 microsecond period (100kHz). This

asymmetry in duty cycle can be easily seen in the Figure 1 waveform.

The test with the Belden 8723 Audio Cable worked if one of the shielded pair was connected to a signal and the other was connected to ground or +5volts. When both wires were connected in parallel as signal wires, the capacitance to ground doubled and the test failed. Also note that the adjacent wire mutual inductive coupling of the SDA and SCL signals did not seem to cause any problems even out to 1000 feet. This indicated that possibly the Belden 9452 45 ohm beldfoil shielded audio cable with a single set of twisted pair wires would be a good candidate to also try.

Flat ribbon cable provided a good compromise between shielding and reasonable capacitance. It is possible to increase the shielding effect by using flat cable with an etched copper foil layer on the back side of the cable. Noise can be induced

into the cable by folding it back over itself for mutual induction effects, and also by operating a noise source close to the cable. A transformer type of soldering iron and florescent light transformers seemed to be good noise sources.

The P82B715 can drive multiple P82B715 remote units. The line should have some form of pull-up resistor at each driver. If only two drivers are used, as shown in Figure 2, the load should be split between the two drivers. For example, if the pull-up current is to be 30 milliamps and the voltage is 5 volts, the pull-up resistance should be: $5V/.030 \text{ amps} = 165 \text{ ohms}$. This should be implemented by placing a 330 ohm resistor at each end of the cable so that the parallel resistance is 165 ohms and each end of the line is terminated. Remembering that the current gain can be as low as 8 and that most runs will not be to the maximum possible distance, lower values of pull-up current can

Using the P82B715 I²C extender on long cables AN444

be used with the appropriate modifications to the above equations.

For larger fan-out with fixed locations, the load resistance should also be evenly divided so that the parallel combination of all of the pull-up resistors will provide the desired D.C. pull-up current.

If some of the remote units will be pluggable, it will be necessary to divide the pull-up load to accommodate all of the possible combinations of possible fanout. Figure 3 shows an example of driving up to 30 remote, pluggable peripherals. On the 3 milliamp side of the P82B715 a complete I²C system may exist. In Figure 3, a local I²C network cluster could be joined to other local network clusters through the P82B715 buffered bus so that hundreds of I²C devices could potentially be interconnected.

The ease of connecting I²C clusters into a complete LAN opens the door for many new uses of components which have an I²C bus connection. Now an electronic instrument can have access to remote keyboards and remote sensors by using the I²C bus. The instrument's output can easily be shown on multiple remote displays all connected with the I²C bus. Multiple instruments can also pass data back and forth over the I²C bus. Thus, we see that the I²C bus can become an effective and inexpensive Local Area Network by using the P82B715 I²C bus extender.

THE TEST SETUP

These tests were run on two identical test boards which each use a Philips Semiconductors 87C751 microcontroller that drives the I²C buffer which has a 330 ohm pull-up resistor. The schematic is shown in Figure 4. The software is a modified version of the "Ping-Pong" program which is described in the Philips Semiconductors Application Note, AN430, "Using the 8XC751/752 in Multimaster applications". This program sends a number down the I²C line and, when received, the receiving unit becomes a master and increments the number and sends it back to the first unit where it is checked and then the process

repeats itself. The software has extensive error detection capability and monitors for corruption of data, false starts, over run of data, stuck lines and about anything else which might indicate a problem. If any errors did occur, a software counter was incremented. In this setup, the counter was stopped at Hex 07F to prevent wrap around and the contents of the counter are displayed on a bank of 8 LEDs. The MSB of the counter register was used as an indicator that the unit was working. The MSB LED flashes at about a 1 Hz rate when the unit is operating normally. When a cable length was reached which was too long, the MSB LED would stop flashing and the counter would rapidly fill up and stop with all 7 LEDs on (LED on indicates a logic "1" in this application).

THE TEST HARDWARE

A general purpose test rig was designed so that future needs of a general I²C platform could also be met. All of the port pins on the '751 were used. The inputs to the system were a toggle switch with a pull-up resistor connected to P0.2 (because this pin is Open Drain) and an octal DIP switch connected to port 1 (the internal pull ups of the port were used, so no external pull-up resistors were needed). The output is displayed through an octal buffer connected to port 3. A logical "1" on the pin will light up the LED. The I²C signals, SDA and SCL, are connected to the I²C buffer chip and the outputs of the buffer are pulled up by 330 ohm resistors. The parallel combination of the buffered transmitting end pull-up and the receiving end pull-up resistors is 330/2 ohms, which results in a pull-up load current of 30 milliamps. This current from the two pull-up resistors must be sunk by the single driving transistor of the acting sender. The effective loading seen by the '751 is the I²C buffer's load divided by 10. Thus, the '751's I²C outputs will sink 3 milliamps when driving the I²C buffer which is sinking 30 milliamps on the buffered bus.

The software monitor routine allows the user to monitor any internal '751 RAM location and display the contents on the LEDs. The monitor routine also allows the user to modify the contents of any RAM location including

SFR space. The Ping-Pong program needed the first 8 locations in RAM, so the stack pointer for this application was changed from the default location of 07H to location 09H. This starts the stack at 0AH.

To read the contents of RAM, set the DIP switches to the desired RAM address. The toggle switch is set to a "1". Pressing the Reset switch causes the microprocessor to reset and then enter the monitor program where the program then waits until the toggle switch is changed. Upon closing the toggle switch (a "1" to "0" transition) the program loads the DIP switch selection into R0 of bank 1 (RAM location 08H). The program then loads the contents of the RAM location pointed to by R0 (bank 1) and copies it into port 3, where it is displayed on the 8 LEDs. Thus, the Address is seen by looking at the DIP switches and the contents pointed to are displayed on the LEDs. Note that this indirect Address latch location (R0, bank 1) would have been the normal beginning of the stack, had it not been changed.

The contents of an internal RAM location can also be modified with this program. First, set the DIP switches to the desired Address and set the toggle switch to "0". Reset the processor and then set the toggle switch to "1". This transfers the address to R0 (bank 1). Next, load the desired new data, which is to be stored in RAM, into the DIP switches, and then set the toggle switch to "0". At this time the LEDs will now show the Address of RAM and the DIP switches show what was written into the selected RAM location. To verify that the data was actually written into the RAM, follow the read RAM sequence.

Although this may seem to be a bit cumbersome, it is a workable way to see what is happening inside of the '751. Remember that it is necessary to re-enter the monitor program, or at least to duplicate the read RAM of R0 (bank 1) and output to port 3, to see the latest version of the contents of the RAM location. Since this experiment only looked at the contents of one RAM location, the above method was easy to use and the display always showed the current status of the desired RAM location because it is updated often by the software.

Using the P82B715 I²C extender on long cables

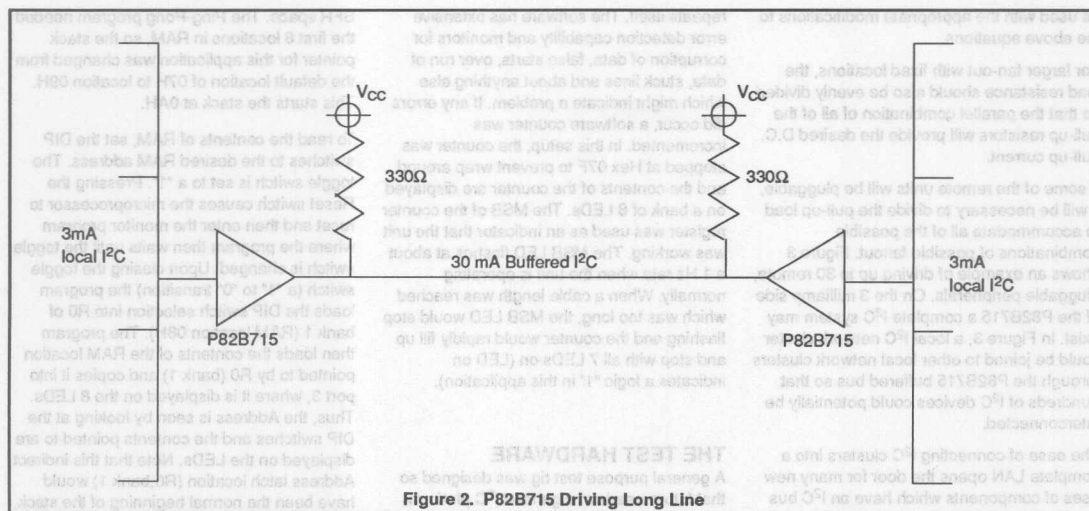
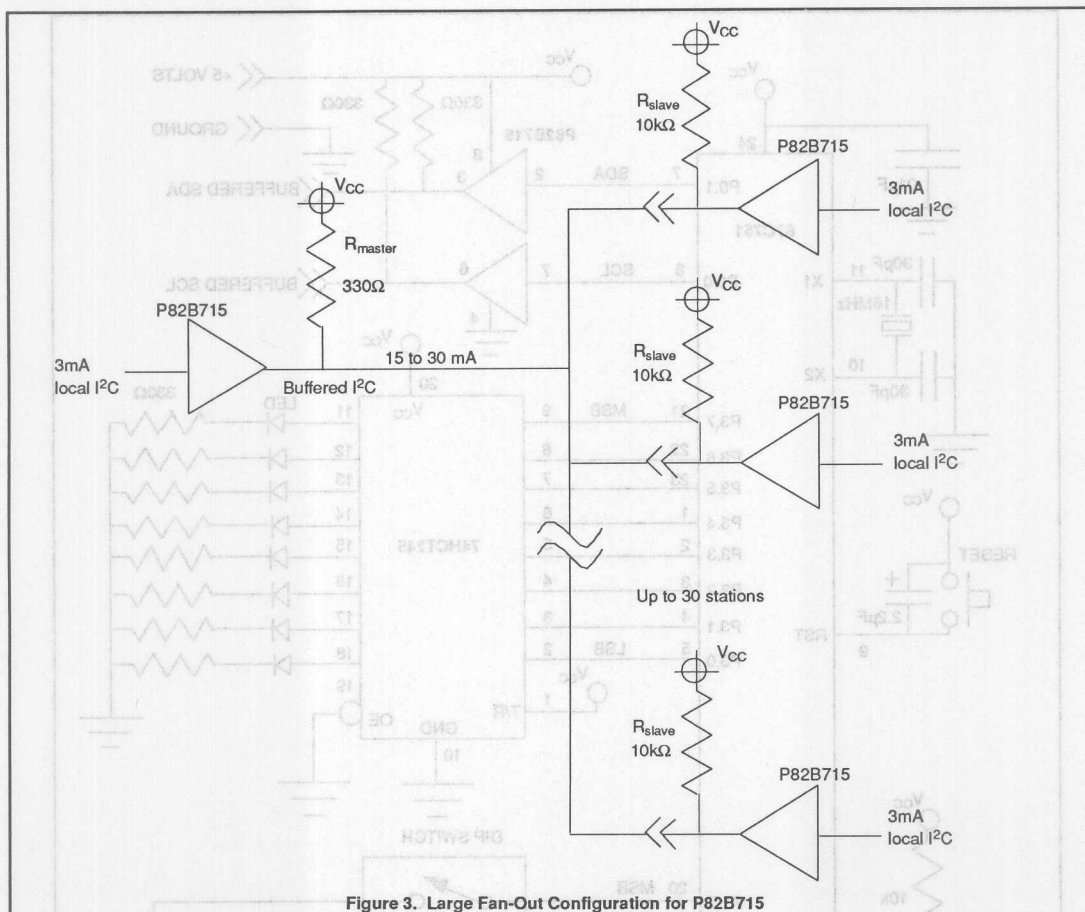


Figure 2. P82B715 Driving Long Line



Note that V_{CC} is 5 volts for these values of load resistors. If a different voltage is desired, the calculations are as follows:

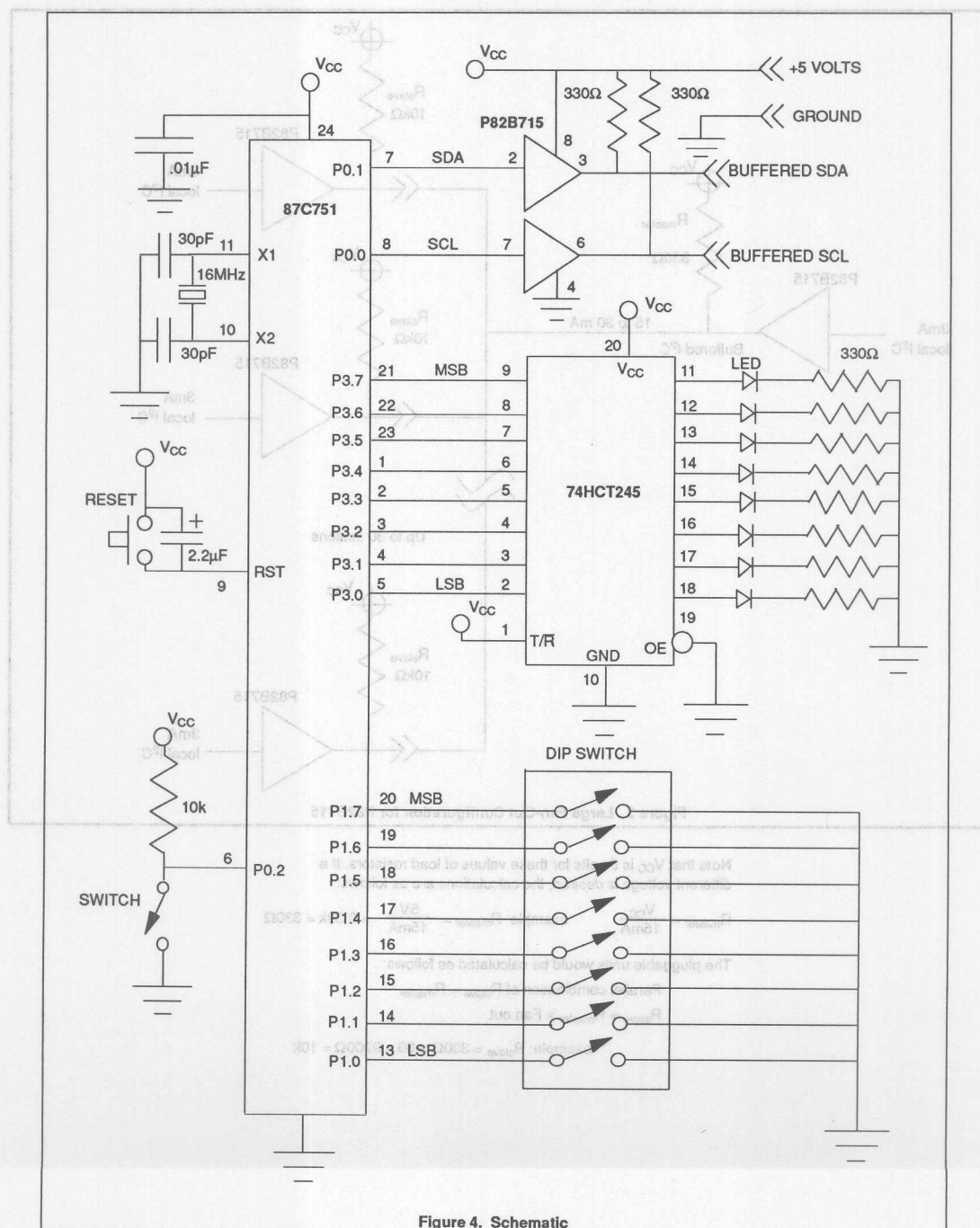
$$R_{\text{master}} = \frac{V_{CC}}{15\text{mA}} \quad \text{example: } R_{\text{master}} = \frac{5\text{V}}{15\text{mA}} = 0.33\text{k} = 330\Omega$$

The pluggable units would be calculated as follows:

Parallel combination of $R_{\text{slave}} = R_{\text{master}}$

$$R_{\text{slave}} = R_{\text{master}} \times \text{Fan out}$$

$$\text{example: } R_{\text{slave}} = 330\Omega \times 30 = 9900\Omega = 10\text{k}$$

Using the P82B715 I²C extender on long cables **AN444**

Using the P82B715 I²C extender on long cables AN444

```

; *****
; Multimaster Code for 83C751/83C752
; 4/14/1992    MODIFIED BY DON SHERMAN 5-21-92
; ;; is used to show where original code was modified
; *****

; This code was written to accompany an application note. The I2C routines
; are intended to be demonstrative and transportable into different
; application scenarios, and were NOT optimized for speed and/or memory
; utilization.

; Yoram Arbel

$TITLE(83C751 Multi Master I2C Routines)
$DATE(4/14/1992)
$MOD751    ;;NEED TO USE $MOD752 FOR 752 EMULATOR
;;EI2      EQU      ES          NEED ENABLE FOR EMULATOR
$DEBBUG

; *****
;      8XC751 MULTIMASTER I2C COMMUNICATIONS ROUTINES
;      Symbols and RAM definitions
; *****

; Symbols (masks) for I2CFG bits.

BTIR      EQU      10h          ; TIRUN bit.
BMRQ      EQU      40h          ; MASTRQ bit.

; Symbols (masks) for I2CON bits.

BCXA      EQU      80h          ; CXA bit.
BIDLE     EQU      40h          ; IDLE bit.
BCDR      EQU      20h          ; CDR bit.
BCARL     EQU      10h          ; CARL bit.
BCSTR     EQU      08h          ; CSTR bit.
BCSTP     EQU      04h          ; CSTP bit.
BXSTR     EQU      02h          ; XSTR bit.
BXSTP     EQU      01h          ; XSTP bit.

; Note:
;
; Specific bits of the I2CON register are set by writing into this register a
; combination of the masks defined above using the MOV command.
; The SETB command should not be used with I2CON, as it is implemented by
; reading the contents of the register, setting the appropriate bit and
; writing it back into the register. As the functionality of the Read and
; Write portions of the I2CON register is different, using SETB may cause
; unwanted results.

; Message transaction status indications in MSGSTAT:

SGO      EQU      10h          ; Started Slave message processing.
SRCVD    EQU      11h          ; as a slave, received a new message
SRLNG    EQU      12h          ; received as slave a message which is too
; long for the buffer
STXED    EQU      13h          ; as slave, completed message transmission.

```

```

SRERR    EQU    14h    ; bus error detected when operating as a slave.

MGO      EQU    20h    ; Started Master message processing.
MRCVED   EQU    21h    ; As Master, received complete message from
                        ; slave.
MTXED    EQU    22h    ; As Master, completed successful message
                        ; transmission (slave acknowledged all data
                        ; bytes).
MTXNAK   EQU    23h    ; As Master, truncated message since slave did
                        ; not acknowledge a data byte.
MTXNOSLV EQU    24h    ; AS Master, did not receive an acknowledgement
                        ; for the specified slave address.

TIMOUT   EQU    30h    ; TIMER1 Timed out.
NOTSTR   EQU    32h    ; Master did not recognize Start.

```

; RAM locations used by I2C interrupt service routines.

```

MASCMD   DATA    20h
SUBADD   BIT      MASCMD.0
RPSTRT   BIT      MASCMD.1
SETMRQ   BIT      MASCMD.2

```

DSEG AT 24h

```

MSGSTAT: DS      1      ; I2C communications status.
MYADDR:  DS      1      ; Address of this I2C node.
DESTADRW: DS      1      ; Destination address + R/W (for Master).
DESSUBAD: DS      1      ; Destination subaddress.
MASTCNT: DS      1      ; Number of data bytes in message (Master,
                        ; send or receive).

TITOCNT: DS      1      ; Timer I bus watchdog timeouts counter.
StackSave: DS      1      ; SP save location (used when returning from
                        ; bus recovery routine).

MasBuf:  DS      4      ; Master receive/transmit buffer, 8 bytes.
SRcvBuf: DS      4      ; Slave receive buffer, 8 bytes.
STxBuf:  DS      4      ; Slave transmit buffer, 8 bytes.

```

```

RBufLen  EQU      4h    ; The length of SRcvBuf

```

; Outputs used by the application:

```

;;TogLED   BIT      P1.0    ; Toggling output pin, to confirm
                        ; that the ping-pong game proceeds fine.
;;ErrLED   BIT      P1.1    ; Error indication.
;;OnLED    BIT      P1.3    ; Indicates an slave message which is too
                        ; long for the buffer.
; Application RAM

```


Using the P82B715 I²C extender on long cables

```

APPFLAGS      DATA      21h
TROFLAG      BIT          APPFLAGS.0
; Flag for monitoring I2C transmission success.
SErrFLAG      BIT          APPFLAGS.1

FAILCNT:      DS          1

TOGCNT:      DS          1          ; Toggle counter.

;*****
;
;                               Program Start
;
;*****
CSEG
; Reset and interrupt vectors.

AJMP          DONMON          ;;JUMP TO MONITOR
;Reset vector at address 0.

;*****
; A timer I timeout usually indicates a 'hung' bus.
;*****

TimerI:      ORG          1Bh          ; Timer I (I2C timeout) interrupt.
              SETB        CLRTI
              AJMP        TIISR          ; Go to Interrupt Service Routine.

;*****
;                               I2C Interrupt Service Routine
;*****
; Notes on the interrupt mechanism:
;
; Other interrupts are enabled during this ISR upon return from XRETI.
; Limitations imposed on other ISR's:
; - Should not be long (close to 1000 clock cycles). A long ISR will cause
;   the I2C bus to 'hang', and a TIMERI interrupt to occur.
; - Other interrupts either do not use the same mechanism for allowing
;   further interrupts, or if they do - disable TIMERI interrupt beforehand.
;
; The 751 hardware allows only one level of interrupts. We simulate an
; additional level by software: by performing a RETI instruction (at location
; XRETI) the interrupt-in-progress flip-flop is cleared, and other interrupts
; are enabled. The second level of interrupt is a must in our implementation,
; enabling timeout interrupts to occur during "stuck" wait loops in the I2C
; interrupt service routine.

ORG          23h

I2CISR:      CLR          EI2          ; Disable I2C interrupt.
              ACALL       XRETI          ; Allow other interrupts to occur.
              PUSH        PSW

```

Using the P82B715 I²C extender on long cables AN444

```

PUSH    ACC
MOV     A,R0
PUSH    ACC
MOV     A,R1
PUSH    ACC
MOV     A,R2
PUSH    ACC

MOV     StackSave, SP
CLR     TIRUN
SETB    TIRUN

JB       STP,NoGo
JNB      MASTER, GoSlave
MOV      MSGSTAT,#MGO
JB       STR,GoMaster
NoGo:    MOV      MSGSTAT,#NOTSTR
AJMP     Dismiss      ; Not a valid Start.

XRETI:   RETI

;*****
;                               Main Transmit and Receive Routines
;*****

; SLAVE CODE -
; GET THE ADDRESS

GoSlave: MOV      MSGSTAT,#SGO
AddrRcv: ACALL    CIsRcv8
JNB      DRDY, SMsgEnd      ; Must be some strange Start or Stop
; before the address byte was completed.
; Not a valid address.

STstRW:  MOV      C,ACC.0
CLR      ACC.0
JZ       GoIdle
; Save R/W~ bit in carry.
; Clear that bit, leaving "raw" address
; If it is a General Address
; - ignore it.

; NOTE:
; One may insert here a different
; treatment for general calls, if
; these are relevant.

JC       SlvTx
; It's a Read - (requesting slave
; transmit).

; It is a Write (slave should receive the message).

; Check if message is for us

SRcv2:   CJNE     A,MYADDR,GoIdle      ; If not my address - ignore the
; message.
MOV      R1,#SRcvBuf
MOV      R2,#RbufLen+1
SJMP     SRcv3

```

Using the P82B715 I²C extender on long cables AN444

```

SRcvSto:  MOV    @R1,A                ; Store the byte
          Inc     R1                  ; Step address.
SRcv3:    ACALL  AckRcv8
          JNB     DRDY,SRcvEnd        ; Exit loop -end reception.
          DJNZ    R2,SRcvSto         ; Go to store byte if buffer not full.

; Too many bytes received - do not acknowledge.
          MOV     MSGSTAT,#SRLNG      ; Notify main that (as slave) we
          ; have received too long a message.
          ACALL  SLnRcvdR            ; Handle new data - slave event routine.
          SJMP   GoIdle

; Received a byte, but not DRDY - check if a legitimate message end.

SRcvEnd:  CJNE    R0,#7,SRcvErr      ; If bit count not 7, it was not
          ; a Start or a Stop.

; Received a complete message

          MOV     MSGSTAT,#SRCVD      ; Calculate number of bytes received
          MOV     A,R1
          CLR     C
          SUBB    A,#SRcvBuf          ; number of bytes in ACC
          ACALL  SRCvdR              ; Handle new data - slave event routine.
          SJMP   SMsgEnd

; It is a Read message, check if for us.

SlvTx:    NOP

STx2:     CJNE    A,MYADDR,GoIdle    ; Not for us.
          MOV     I2DAT,#0           ; Acknowledge the address.
          JNB     ATN,$              ; Wait for attention flag.
          JNB     DRDY,SMsgEnd       ; Exception - unexpected Start
          ; or Stop before the Ack got out.
          MOV     R1,#STxBuf         ; Start address of transmit buffer.
STxlp:    MOV     A,@R1              ; Get byte from buffer
          INC     R1
          ACALL  XmByte              ; Form a write bit with address.
          JNB     DRDY,SMsgEnd       ; Byte Tx not completed.
          JNB     RDAT,STxlp         ; Byte acknowledge, proceed trans.
          MOV     I2CON,#BCDR+BIDLE ; Master Nak'ed for msg end.
          MOV     MSGSTAT,#STXED     ; The address.
          ACALL  STXedR              ; Slave transmitted event routine.
          AJMP   Dismiss

SRcvErr:  MOV     MSGSTAT,#SRERR      ; Flag bus/protocol error
          ACALL  SRErrR              ; Slave error event routine.
          SJMP   SMsgEnd

StxErr:   MOV     MSGSTAT,#SRERR      ; Flag bus/protocol error
          ACALL  SRErrR

```

```

SMsgEnd:  JB    MASTER,SMsgEnd2      ; Start the byte
          JB    STR,GoSlave          ; If it was a Start, be Slave
SMsgEnd2:  AJMP  Dismiss              ; Exit loop and reception
                                         ; Go to store byte in buffer and full

; End of Slave message processing

GoIdle:    AJMP  Dismiss

; Received a byte, but not R/W~ - check if a legitimate message end
; If bit count not 7, it was not
; a Start or a Stop

GoMaster:

; Send address & R/W~ byte
          MOV    R1,#MasBuf          ; Master buffer address
          MOV    R2,MASTCNT          ; # of bytes, to send or rcv
          MOV    A,DESTADRW          ; Destination address (including
                                         ; R/W~ byte).
          JB     SUBADD,GoMas2        ; Branch if subaddress is needed.

          ACALL  XmAddr

          JNB    DRDY,GM2
          JNB    ARL,GM3
GM2:       AJMP  AdTxAr1              ; Arbitration loss while transmitting
                                         ; the address.
GM3:       JB    RDAT,Noslave         ; No Ack for address transmission.
          JB     ACC.0, MRcv          ; Check R/W~ bit
          AJMP  MTx

; Handling subaddress case:
GoMas2:    NOP                      ; Subaddress needed. Address in ACC.
          CLR    ACC.0               ; Force a Write bit with address.
          ACALL  XmAddr
          JNB    DRDY,GM4
          JNB    ARL,GM5
GM4:       AJMP  AdTxAr1              ; Arbitration loss while transmitting
                                         ; the address.
GM5:       JB    RDAT,Noslave         ; No Ack for address transmission.
          MOV    A,DESSUBAD          ; Transmit subaddress.
          ACALL  XmByte
          JNB    DRDY,SMsgEnd2        ; Arbitration loss (by Start or Stop)
          JB     ARL,SMsgEnd2        ; Arbitration loss occurred.
          JB     RDAT,NoAck          ; Subaddress transmission was not ack'ed.
          MOV    A,DESTADRW          ; Reload ACC with address.
          JNB    ACC.0, MTx          ; It's a Write, so proceed
                                         ; by sending the data.

```

Using the P82B715 I²C extender on long cables AN444

```

; Read message, needs rp. Start and add. retransmit.
; Consider this Master message
; Send Stop, or a Repeated Start

MOV     I2CON,#BCDR+BXSTR ; Send Repeated Start.
JNB     ATN,$
MOV     I2CON,#BCDR
; Clear useless DRDY while preparing
; for Repeated Start.
JNB     ATN,$
; expecting an STR.
JNB     ARL,GM6
AJMP    MarlEnd
; oops - lost arbitration.
GM6:    ACALL  XmAddr ; Retransmit address, this time with the
; Read bit set.
JNB     DRDY,GM7
JNB     ARL,GM8
GM7:    AJMP    AdTxAr1 ; Arbitration loss while transmitting
; the address.
GM8:    JB      RDAT,Noslave ; No Ack - the slave disappeared.
SJMP    MRcv ; Proceed receiving slave's data.

; A Write message. Master transmits the data.

MTx:     NOP

MTxLoop: MOV     A,@R1 ; Get byte from buffer.
INC      R1 ; Step the address.
ACALL    XmByte
JNB     DRDY,SMsgeEnd2 ; Arbitration loss (by Start or Stop)
JB      ARL,SMsgeEnd2 ; Arbitration loss.
JB      RDAT,NoAck
DJNZ    R2,MTxLoop ; Loop if more bytes to send.

MOV     MSGSTAT,#MTXED ; Report completion of buffer
; transmission.

NoSlave: SJMP    MTxStop
MOV     MSGSTAT,#MTXNOSLV
NoAck:   SJMP    MTxStop
MOV     MSGSTAT,#MTXNAK
SJMP    MTxStop

; Master receive - a Read frame

MRcv:    ACALL    ClaRcv8 ; Receive a byte.
SJMP    MRcv2
MRcvLoop: ACALL    AckRcv8
MRcv2:   JNB     DRDY,Marl ; Other's Start or Stop.
MOV     @R1,A ; Store received byte.
INC     R1 ; Advance address.
DJNZ    R2,MRcvLoop

; Received the desired number of bytes - send Nack.

MOV     I2DAT,#80h
JNB     ATN,$
JNB     DRDY,Marl
MOV     MSGSTAT,#MRCVED
SJMP    MTxStop ; Go to send Stop or Repeated Start.

```


Using the P82B715 I²C extender on long cables AN444

```
; Conclude this Master message:
; Send Stop, or a Repeated Start
```

```
MTxStop:  JNB      RPSTRT,MTxStop2      ; Check if Repeated Start needed
                                                ; Around if not RPSTRT.
MOV        I2CON,#BCDR+BXSTR           ; Send Repeated Start.
SJMP       MTxStop3
MTxStop2:  MOV      C,SETMRQ             ; Set new Master Request if demanded
MOV        MASTRQ,C                     ; by SETMRQ bit of MASCMD.
MOV        I2CON,#BCDR+BXSTP           ; Request the HW to send a Stop.

MTxStop3:  JNB      ATN,$                ; Wait for Attention
MOV        I2CON,#BCDR                 ; Clear the useless DRDY, generated
                                                ; by SCL going high in preparation
                                                ; for the Stop or Repeated Start.
JNB        ATN,$                        ; Wait for ARL, STP or STR.
JB         ARL,MarlEnd                  ; Lost arbitration trying to send
                                                ; Stop or a ReStart.
```

```
; Master is done with this message. May proceed with new messages, if any,
; or exit.
```

```
ACALL      MastNext                     ; Master Event Routine. May Prepare
                                                ; the pointers and data for the
                                                ; next Master message.

JNB        MASTRQ,MMsgEnd               ; Go end service routine if MASTRQ
                                                ; does not indicate that the master
                                                ; should continue (was set according
                                                ; to SETMRQ bit, or by MastNext).

JNB        STR,MMsgEnd                  ; Return from the ISR, unless Start
                                                ; (avoid danger if we do not return:
                                                ; if there was a Stop, the watchdog
                                                ; is inactive until next Start).
AJMP       GoMaster                     ; Loop for another Master message
;
MMsgEnd:   SJMP      Dismiss             ; End of Master messages,
```

```
; Terminate mastership due to an arbitration loss:
```

```
Marl:
```

```
JNB        STR,Marl2                   ; If lost arbitration due to other
AJMP       GoSlave                      ; Master's Start, go be a slave.
```

```
Marl2:
```

```
AJMP      Dismiss
```

```

; Switch from Master to Slave due to arbitration loss after completing
; transmission of a message. The MASTRQ bit was cleared trying to write a
; Stop, and we need to set it again on order to retry transmission when the
; bus gets free again.

```

MarlEnd:

```

    SETB    MASTRQ        ; Set Master Request - which will get
                          ; into effect when we are done as a
                          ; slave.
    ACALL   MORERR        ; INCREASE ERROR COUNT
    AJMP    Marl

```

; Handling arbitration loss while transmitting an address

```

AdTxAr1:  JB     STR,Marl    ; Non-synchronous Start or Stop.
          JB     STP,Marl

```

```

; Switch from Master to Slave due to arbitration loss while transmitting
; an address - complete receiving the address transmitted by the new Master.

```

```

    CJNE    R0,#0,AdTxAr12  ; Ar1 on last bit of address
                          ; (R0 is 0 on exit from XmAddr).
    DEC     A                ; The lsb sent, in which ar1 occurred
                          ; must have been 1. By decrementing
                          ; A we get the address that won.
    SJMP    AdAr3

AdTxAr12:
    RR      A                ; Realign partially Tx'ed ACC
    MOV     R1,A             ; and save it in R1
    MOV     A,R0             ; Pointer for lookup table
    MOV     DPTR,#MaskTable
    MOVC    A,@A+DPTR
    ANL     A,R1             ; Set address bits to be received,
                          ; and the bit on which we lost
                          ; arbitration to 0
                          ; Now we are ready to receive the rest
                          ; of the address.

```

```

    MOV     I2CON,#BCXA+BCARL ; Clear flags and release the clock.

```

```

    ACALL   RBit3           ; Complete the address using reception
                          ; subroutine.
    JB      DRDY,AdAr3      ; Around if received address OK
    AJMP    SMsgEnd         ; Unexpected Start or Stop - end
                          ; as a slave.
AdAr3:     AJMP    STstRW    ; Proceed to check the address
                          ; as a slave.

```

```

MaskTable: DB      0ffh,7Eh,3Eh,1Eh,0Eh,06h,02h,00h, ; 0ffh is dummy

```

; End I2C Interrupt Service Routine:

```

Dismiss:  ACALL   I2CDONE

```

Using the P82B715 I²C extender on long cables AN444

```

MOV     I2CON, #BCARL+BCSTP+BCDR+BCXA+BDLE
CLR     TIRUN
POP     ACC
MOV     R2, A
POP     ACC
MOV     R1, A
POP     ACC
MOV     R0, A
POP     ACC
POP     PSW
SETB    EI2

; Return from I2C interrupt Service Routine
RET

; Handling arbitration loss while transmitting an address
; *****
;
; Byte Transmit and Receive Subroutines
; *****
;
; Switch from Master to Slave due to arbitration loss while transmitting
; an address - complete receiving the address transmitted by the new
; master. Transmit a byte
; *****
;
; Set on last bit of address
;
XmAddr: MOV     I2DAT, A           ; Send first bit, clears DRDY.
        MOV     I2CON, #BCARL+BCSTR+BCSTP
        ; Clear status, release SCL.
        MOV     R0, #8
        SJMP    XmBit2
XmByte: MOV     R0, #8
XmBit:  MOV     I2DAT, A           ; Send the first bit.
XmBit2: RL      A                 ; Get next bit.
        JNB     ATN, $            ; Wait for bit sent.
        JNB     DRDY, XmBex       ; Should be data ready.
        DJNZ    R0, XmBit         ; Repeat until all bits sent.
        MOV     I2CON, #BCDR+BCXA ; Switch to receive mode.
        JNB     ATN, $            ; Wait for acknowledge bit.
        ; flag cleared.
XmBex:  RET

;
; Byte receive routines.
;
; CIsRcv8 clears the status register (from Start condition)
; and then receives a byte.
; AckRcv8 Sends an acknowledge, and then receives a new byte.
; If a Start or Stop is encountered immediately after the
; ack, AckRcv8 returns with 7 in R0.
; ClaRcv8 clears the transmit active state and releases clock
; (from the acknowledge).
;
; A contains the received byte upon return.
; R0 is being used as a bit counter.
;
; End I2C Interrupt Service Routine
; *****

CIsRcv8: MOV     I2CON, #BCARL+BCSTR+BCSTP+BCXA
        JNB     ATN, $

```

Using the P82B715 I²C extender on long cables AN444

```

JNB     DRDY,RCVex
SJMP    Rcv8

AckRcv8: MOV     I2DAT,#0           ; Send Ack (low)
JNB     ATN,$
JNB     DRDY,RCVerr                ; Bus exception - exit.
ClaRcv8: MOV     I2CON,#BCDR+BCXA   ; clear status, release clock
                                           ;from writing the Ack.
JNB     ATN,$

Rcv8:   MOV     R0,#7               ; Set bit counter for the first seven
                                           ; bits.
CLR     A                          ; Init received byte to 0.
RBit:   ORL     A,I2DAT              ; Get bit, clear ATN.
RBit2:  RL      A                   ; Shift data.
JNB     ATN,$                       ; Wait for next bit.
JNB     DRDY,RCVex                  ; Exit if not a data bit (could be Start/
                                           ; Stop, or bus/protocol error)
RBit3:  DJNZ    R0,RBit              ; Repeat until 7 bits are in.
MOV     C,RDAT                      ; Get last bit, don't clear ATN.
RLC     A                           ; Form full data byte.
RCVex:  RET

RCVerr: MOV     R0,#9               ; Return non legitimate bit count
RET

;*****
;       Timer I Interrupt Service Routine
;       I2C us Timeout
;*****
; In addition to reporting the timeout in MSGSTAT, we update a failure
; counter, TI0CNT. This allows different types of timeout handling by the
; main program.

TIISR:  CLR     MASTRQ               ; "Manual" reset.
MOV     I2CON,#BXSTP                ;
MOV     I2CON,#BCXA+BCDR+BCARL+BCSTR+BCSTP

TI1:    MOV     MSGSTAT,#TIMOUT      ; Status Flag for Main.
TI2:    ACALL   MORERR               ;;INC TI0CNT
TI4:    ACALL   RECOVER

SETB    CLRTI                       ; Clear TI interrupt flag.
ACALL   XRETI                       ; Clear interrupt pending flag (in
                                           ; order to re-enable interrupts).
MOV     SP,StackSave                ; Realign stack pointer, re-doing
                                           ; possible stack changes during
                                           ; the I2C interrupt service routine.
                                           ; TimerI interrupts in other ISR's
                                           ; were not allowed !
AJMP    Dismiss                     ; Go back to the I2C service routine,
                                           ; in order to return to the (main)
                                           ; program interrupted.
;*****

```

Using the P82B715 I²C extender on long cables AN444

```

; Bus recovery attempt subroutine
;*****

RECOVER: CLR EA
CLR MASTRQ ; "Manual" reset.
MOV I2CON, #BCXA+BDLE+BCDR+BCARL+BCSTR+BCSTP
CLR SLAVEN ; Non I2C TimerI mode
SETB TIRUN ; Fire up TimerI. When it overflows, it
; will cause I2C interface hardware reset.

DLY5: MOV R1, #0ffh
NOP
NOP
NOP
DJNZ R1, DLY5
CLR TIRUN
SETB CLRTI

SETB SCL ; Issue clocks to help release other devices.
SETB SDA
MOV R1, #08h
RC7: CLR SCL
DB 0,0,0,0,0
SETB SCL
DB 0,0,0,0,0
DJNZ R1, RC7
CLR SCL
DB 0,0
CLR SDA
DB 0,0
SETB SCL
DB 0,0,0,0,0
SETB SDA
DB 0,0,0,0,0 ; Issue a Stop.

Rex: MOV I2CON, #BCXA+BCDR+BCARL+BCSTR+BCSTP ; clear flags
SETB EA
RET

;*****
;
; Main Program
;*****

; Message ping pong game. Each message is transmitted by
; a processor that is a master on the I2C bus, and it contains one byte
; of data. A processor that receives this data byte as a slave increments
; the data by one and transmits it back as a master. The data received is
; confirmed to be a one increment of the data formerly sent, unless
; it is a "reset" value, chosen to be 00h.
; The two participating processors have similar code, where the node
; address of the second processor is the destination address of this
; one, and vice versa.
; The first data byte each processor tries to send is 00h. One of the
; processors will acquire the bus first, and the second processor that will
; receive this "resetting" 00h will not attempt to confirm it against an
; expected value. It will simply increment and transmit it. Subsequent
; receptions will be confirmed against the expected value, until 0ffh data

```


Using the P82B715 I²C extender on long cables AN444

```

; bytes are sent and the game is effectively reset by the 00h resulting from
; the next increment.
; A toggling output (TogLED) tells the outer world that the "ping pong"
; proceeds well. If something unexpected happens we temporarily activate
; another output, ErrLED.
; The different tasks of the code are performed in a combination of main-
; line program and event routines called from the I2C interrupt service
; routine.

; Initial set-ups:
;   Load CT1,CT0 bits of I2CFG register, according to the clock
;   crystal used.
;   Load RAM location MYADDR with the I2C address of this processor.
;   We load these values out of ROM table locations (R_CTVAL and R_MYADDR).
;   One may, instead, load with a MOV <immediate> command.

;;Reset:    MOV    SP,#07h          ;Set stack location.
RESET:      CLR    A
            MOV    DPTR,#R_CTVAL
            MOVC   A,@A+DPTR
            MOV    I2CFG,A          ; Load CT1,CT0 (I2C timing, crystal
                                     ; dependent).
            CLR    A
            MOV    DPTR,#R_MYADDR
            MOVC   A,@A+DPTR        ; Get this node's address from ROM table
            MOV    MYADDR,A         ; into MYADDR RAM location.

;;          CLR    OnLED

;;Reset2:   CLR    ErrLED          ; Flash LED.
RESET2:     ACALL  LDELAY
;;          SETB   ErrLED
            CLR    SErrFLAG
            CLR    TRQFLAG
            MOV    FAILCNT,#50h
;;          SETB   TogLED
            MOV    TOGCNT,#050h    ; Initialize pin-toggling counter

; Enable slave operation.
; The Idle bit is set here for a restart situation - in normal
; operation this is redundant, as this bit is set upon power up reset.
            MOV    I2CON,#BIDLE    ; Slave will idle till next Start.
            SETB   SLAVEN          ; Enable slave operation.

; Enable interrupts.
; This is necessary for both Slave and Master operations.
            SETB   ETI             ; Enable timer I interrupts.
            SETB   EI2            ; Enable I2C port interrupts.
            SETB   EA             ; Enable global interrupts.

; Set up Master operation.
            MOV    MASCMD,#0h      ; "Regular" master transmissions.
            MOV    DPTR,#PongADDR
            CLR    A
            MOVC   A,@A+DPTR

```

Using the P82B715 I²C extender on long cables AN444

```

MOV     DESTADRW,A      ; The partner address. The LSB is
                        ; low, for a Write transaction.
MOV     MASTCNT,#01h    ; Message length -- a single byte.

PPSTART:
MOV     MasBuf,#00h

; "Ping" transmission:

PP2:
        SETB    TRQFLAG
        SETB    MASTRQ
        MOV     R1,#0ffh
PP22:   JNB     TRQFLAG,PP3      ; Transmitted OK
        DJNZ    R1,PP22
MFAIL1: DJNZ    FAILCNT,PP2
        ACALL   MORERR          ; INCREMENT TITOCNT
        ACALL   RECOVER
        SJMP    Reset2

; "Pong" reception:

PP3:    MOV     R0,#0ffh      ; Software timeout loop count.
PP31:   MOV     R1,#0ffh
PP32:   JB      TRQFLAG,PP2    ; Rcvd ok as slave, go transmit.
        JB      SErrFLAG,PP5
        DJNZ    R1,PP32
        DJNZ    R0,PP31
PPTO:   ACALL   RECOVER        ; Software timeout.
        AJMP    Reset2

;;PP5:   CLR     ErrLED        ; Receive error.
;;      ACALL   LDELAY
;;      SETB    ErrLED
PP5:    CLR     SErrFLAG
        AJMP    PPSTART

LDELAY:  MOV     R2,#030h      ; LONG DELAY
LDELAY1: MOV     R1,#0ffh
        DJNZ    R1,$
        DJNZ    R2,LDELAY1
        RET

;*****
; Slave and Master Event Routines.
;*****

;
; Invoked upon completion of a message transaction.
; This is the part of the application program actually dealing
; with the data communicated on the I2C bus, by responding to
; new data received and/or preparing the next transaction.

; Slave Event Routines
;
; These routines are invoked by the I2C interrupt service routine when a
; message transaction as a slave has been completed. Our "application"

```

Using the P82B715 I²C extender on long cables

```

; reacts to a message received as a slave with the routine SRCvdR.
; The calls that indicate erroneous reception are treated the same way as
; erroneous data reception in the "ping pong" game.

; SRCvdR
; Invoked when a new message has been received as a Slave.

SRCvdR:  NOP
        MOV     A,SRcvBuf
        JNZ     SR2
        MOV     MasBuf,#01h      ; It was ping-pong reset value
        SJMP    SR3

SR2:     INC     MasBuf           ; The expected data.
        CJNE    A,MasBuf,ErrSR
        INC     MasBuf           ; Data for next transmission - the data
                                   ; received incremented by 1.

; A successful two way data exchange. Let the outside world know by
; toggling an output pin driving a LED. We actually toggle only
; when a number of such exchanges is completed, in order to
; slow down the changes for a good visual indication.

        DJNZ    TOGCNT,SR3
        CPL     TogLED           ; Toggle output
        XRL     TITOCNT, #80H    ;;TOGGLE MSB LED
        MOV     TOGCNT,#050h
        SETB    PSW.3            ;;RS TO 1
        MOV     LED, @R0         ;;RAM POINTED TO BY R0
        CLR     PSW.3            ;;RS BACK TO 0
SR3:     CLR     SErrFLAG
        SETB    TRQFLAG          ; Request main to transmit
        RET

ErrSR:   SETB    SErrFLAG
        RET

; SLnRcvdR
; Invoked when a message received as a Slave is too long
; for the receive buffer.

; STXedR
; Invoked when a Slave completed transmission of its buffer.
; We do not expect to get here, since we do not plan to have
; in our system a master that will request data from this node.

; SRErrR
; Slave error event subroutine.
; In most applications it will not be used.

SLnRcvdR:
STXedR:
SRErrR:  JMP     ErrSR

```

Using the P82B715 I²C extender on long cables AN444

```

;
; MastNext - Master Event Routine.
;
; Invoked when a Master transaction is completed, or terminated
; "willingly" due to lack of acknowledge by a slave.
;
; Invoked when a new message has been received as a slave.

MastNext:
    MOV     A,MSGSTAT
    CJNE    A,#MTXED,MN1
    MOV     FAILCNT,#50h
    CLR     TRQFLAG
    RET

MN1:
    RET

; I2CDONE
; Called upon completion of the I2C interrupt service routine.
; In this example it monitors exceptions, and invokes the bus
; recovery routine when too many occurred.

I2CDONE:
    MOV     A,MSGSTAT
    CJNE    A,#NOTSTR,I2CD1
    ACALL   MORERR
    DJNZ    FAILCNT,I2CD1
    MOV     FAILCNT,#01h
    CLR     EI2
    RET

I2CD1:
    RET

;*****
; I2C Communications Table:
;*****

; We used table driven values for clarity. One may use immediates to load
; these values and save several lines of code.

; Contents is used in the beginning of the main program to load
; RAM location MYADDR and the I2CFG register.
; The node address, in R_MYADDR, is application specific, and unique for
; each device in the I2C network.
; R_CTVAL depends on the crystal clock frequency.

R_MYADDR:  DB      4Ah          ; This node's address
; NOTE THAT R_MYADDR AND PongADDR
; MUST BE SWITCHED ON THE OTHER
; '751

R_CTVAL:   DB      02h          ; CT1, CT0 bit values

;*****
; Application Code Definitions
;*****

PongADDR:  DB      4Eh          ; The address of the "partner" in
; the ping-pong game.

```

Using the P82B715 I²C extender on long cables AN444

```

;;I2CMON      THIS PROGRAM RUNS THE MONITOR ON
;;           THE SMALL TEST BOARD DESIGNED TO
;;           TEST THE I2C DRIVER CHIP.
;;           IT USES A '751.
;
;
;
LED      EQU    P3
LDEL     EQU    022H
HDEL     EQU    LDEL + 1
SWITCH   EQU    P1
TOG       EQU    P0.2      ;TOGGLE SWITCH
RNAME    EQU    R0        ;R0 RAM POINTER
;
;
;
DONMON:  MOV     SP,      #09H      ;SP=09, STARTS AT 0AH
         SETB    PSW.3      ;RS = 01
         CLR     PSW.1      ;PSW.1 FLAG=0
         JB      TOG,      ONLYAD   ;IF TOG 1, PSW1=0
         SETB    PSW.1      ;WRITE DESIRED
ONLYAD:   JNB     TOG,      ONLYAD   ;WAIT FOR HI
HIWAIT:   JB      TOG,      HIWAIT   ;NOW WAIT FOR LOW
         MOV     LDEL,     #0      ;DELAY TIMER
         MOV     HDEL,     #0
SDELAY:   DJNZ    LDEL,     SDELAY   ;DELAY LOOP
         DJNZ    HDEL,     SDELAY   ;UPPER DELAY
         JB      TOG,      HIWAIT   ;FALSE ALARM, GO BACK
         MOV     RNAME,    SWITCH   ;VALID HI TO LO
         MOV     LED,      @RNAME   ;DISPLAY CONTENTS OF
         ;        RAM OF RNAME
         JNB     PSW.1,     DONE     ;PSW1 FLAG, 0=DONE
         TOG     STAYLO    ;NOW WAIT FOR HI
HDELAY:   DJNZ    LDEL,     HDELAY   ;LDEL=HDEL=0
         DJNZ    HDEL,     HDELAY
         TOG     STAYLO    ;FALSE ALARM
         MOV     @RNAME,    SWITCH   ;SUCCESSFUL LO TO HI
         ;        SWITCH TO RAM
         MOV     LED,      RNAME    ;DISPLAY WHICH RAM
         ;        LOCATION FOR SWITCH
DONE:     CLR     PSW.3      ;RS BANK BACK TO 0
         AJMP    RESET      ;STARTS PING PONG
;
;
;
MORERR:   PUSH    ACC
         MOV     A,        #7FH    ;INCREMENT TITOCNT
         ANL     A,        TITOCNT
         XRL     A,        #7FH    ;STOP AT 7F
         JZ      NOUP
         INC     TITOCNT
         SETB    PSW.3      ;RS TO 1
         MOV     LED,      @R0     ;DISPLAY NEW TITOCNT
         CLR     PSW.3      ;RS BACK TO 0
NOUP:     POP     ACC
         RET
;
END

```


PLM51 I²C software interface IIC51 (version 0.5) ETV/AN89004

Author: R.C.J. Brink, Eindhoven

1. INTRODUCTION.

1.1. Purpose.

This document is a user manual for the I²C software module IIC51. It is intended for Intel PLM51 users who need to control an I²C bus. This document assumes some basic knowledge about I²C and Intel PLM51.

1.2. Scope.

IIC51 is a software module to provide an Intel PLM51 user with a set of procedures to control a bi-directional I²C bus. These procedures have been coded in Intel ASM51 and have been optimized for speed. IIC51 supports all common used I²C master transmitter and master receiver protocols. Each different protocol corresponds to one of the procedures in IIC51. IIC51 is available in two different versions:

IIC51S:

IIC51S is a module for singlemaster I²C to be used on microcontrollers of the 8x51 family. It directly controls the microcontroller I/O pins by software without the need of any specific hardware. No other I²C masters are allowed on the bus. Note that the electrical characteristics of this microcontroller family are not conform the I²C specifications.

IIC51M:

IIC51M is a module for multimaster I²C to be used on microcontrollers of the PCB8xC552/C652 family. It makes use of the built-in I²C interface hardware (SIO1) of these microcontrollers. Since this hardware is a multimaster interface other I²C masters are allowed on the bus.

All I²C transfer procedures in IIC51S are fully software interface compatible with IIC51M. This allows a single PLM51 program using I²C to be written for both mentioned microcontroller families.

1.3. Definitions, Acronyms and Abbreviations.

I ² C	Inter-IC bus
PLM51	High level Program Language for 8051 family Microcontrollers
ASM51	Assembly Language for 8051 family Microcontrollers
RL51	Relocating Linker for 8051 family Microcontrollers
S	I ² C Message Start Condition
P	I ² C Message Stop Condition
A	I ² C Message Acknowledge
N	I ² C Message Negative Acknowledge
SlvW	I ² C Message Slave Address + Write
SlvR	I ² C Message Slave Address + Read
Sub	I ² C Slave Subaddress

NV-Memory I²C Controlled Non Volatile Memory (E²PROM)

PLM51 I²C software interface IIC51 (version 0.5) ETV/AN89004

1.4. References.

- | | | |
|-----|---|------|
| [1] | I ² C Specification
I ² C-bus compatible ICs
Philips Components Data Handbook IC12a | 1989 |
| [2] | PL/M-51 User's Guide for DOS Systems
Intel Corporation | 1986 |
| [3] | MSC-51 Macro Assembler User's Guide for DOS Systems
Intel Corporation | 1986 |
| [4] | MSC-51 Utilities User's Guide for DOS Systems
Intel Corporation | 1986 |
| [5] | Single-chip 8-bit microcontrollers PCB83C552/PCB80C552, PCB83C652/PCB80C652 etc.
I ² C-bus compatible ICs
Philips Components Data Handbook IC12a | 1989 |
| [6] | Single-chip 8-bit microcontroller PCB80C51
Integrated circuits Book IC14 | 1987 |

PLM51 I²C software interface IIC51 (version 0.5) ETV/AN89004

2. GENERAL DESCRIPTION.

2.1. Perspective.

IIC51 is designed for use in stand-alone microcontroller I²C systems. It is mainly written to provide a standard set of procedures for computer controlled television / teletext concepts based on 8051 family microcontrollers.

2.2. Functions.

IIC51 contains the following functions:

- Initialisation of the I²C interface (software and hardware)
- Transfer of I²C messages to and from an I²C slave device
- Error detection
- Automatic retrying if an error occurs during a transfer (up to 5 attempts)
- Error recovery if the bus is held by a slave device that is out of bit-sync
- Optional slave receiver / transmitter function (IIC51M only)

2.3. User Characteristics.

IIC51 is designed to be a easy to use package. All needed code and data is defined in a single object module (IIC51M.OBJ or IIC51S.OBJ). The PLM51 user needs only to link this module to his own application object modules, using Intel's RL51. Procedures and data of concern to the user can be declared EXTERNAL by including the file IIC51.DCL.

2.4. General Constraints.

IIC51 is coded for and translated by the Intel MSC-51 Macro Assembler. It is tested together with Intel PLM51 modules. Intel utilities used for testing:

- MSC-51 Macro Assembler, ASM51.EXE, Version V2.3
- PL/M-51 Compiler, PLM51.EXE, Version V1.2 and V1.3
- MSC-51 Relocator and Linker, RL51.EXE, Version V3.1

Development is done on an IBM-PC/AT running DOS.

IIC51S needs:

- 350 Bytes CODE (approx.)
- 6 Bytes DATA
- 1 Byte Bit-Addressable DATA
- 1 Bit

IIC51M needs:

- 400 Bytes CODE (approx.)
- 6 Bytes DATA
- 1 Byte Bit-Addressable DATA
- 1 Bit
- Exclusive use of Register Bank 1

PLM51 I²C software interface IIC51 (version 0.5) ETV/AN89004

3. FUNCTIONAL DESCRIPTION.

3.1. Master Mode Functions.

This section describes the available functions in IIC51 on a procedure by procedure bases.

Each procedure must be declared EXTERNAL by the PLM51 user. In this declaration the user can specify the type returned by each procedure. All procedures (except Init_IIC) can return a BIT or a BYTE (depending on the chosen EXTERNAL declaration). The BIT or BYTE returned is 0 if the I²C transmission was successful. If the user decides to declare a procedure untyped, the result of the previous I²C transmission can always be checked by examining the static BIT variable IIC_Error. Note that typed procedures must be called using an expression. If the result of an I²C procedure is to be ignored, a dummy assignment must be done for a typed procedure. An untyped procedure can be called by the PLM51 CALL statement, without any additional overhead. The examples in the follow section assume the procedures to be declared untyped.

Note that the least significant bit of all slaveaddresses passed to the I²C procedures must be 0.

3.1.1. Init_IIC.

Declaration:

```
Init_IIC:
  PROCEDURE ( Own_Slave_Address ) EXTERNAL ;
  DECLARE   ( Own_Slave_Address ) BYTE ;
  END ;
```

Description:

Init_IIC must be called once after reset, before any other procedure is used. It initialises all I²C internal static data and hardware. The Own_Slave_Address is passed to Init_IIC for the optional slave function in a multimaster I²C system (IIC51M). In a singlemaster I²C system (IIC51S), the Own_Slave_Address is ignored. Note that Init_IIC does not effect the global interrupt enable flag (EA). IIC51M requires the user to enable interrupts afterwards (see example).

Example:

```
CALL Init_IIC ( 54h ) ;
ENABLE ;           /* Enable Interrupts; EA = 1 */
```

PLM51 I²C software interface IIC51 (version 0.5) ETV/AN89004

3.1.2. IIC_Test_Device.

Declaration:

```

IIC_Test_Device:
  PROCEDURE ( Slave_Address ) [ BIT | BYTE ] EXTERNAL ;
  DECLARE ( Slave_Address ) BYTE ;
  END ;

```

Description:

IIC_Test_Device just sends the slaveaddress on the I²C bus. It can be used to check the presence of a device on the I²C bus.

I²C Protocol:

S	SlvW	A	P
---	------	---	---

(Device is Present, IIC_Error = 0)

OR

S	SlvW	N	P
---	------	---	---

(Device is Not Present, IIC_Error = 1)

Example:

```

DECLARE IIC_Error BIT EXTERNAL ;
.....
CALL IIC_Test_Device ( 8Ch ) ;
IF ( IIC_Error ) THEN
  "Device is Not Present Handling"
ELSE
  "Device is Present Handling"

```


PLM51 I²C software interface IIC51 (version 0.5) ETV/AN89004

3.1.3. IIC_Write.

Declaration:

```

IIC_Write:
  PROCEDURE ( Slave_Address, Count, Source_Ptr ) [ BIT | BYTE ] EXTERNAL ;
  DECLARE   ( Slave_Address, Count, Source_Ptr ) BYTE ;
  END ;

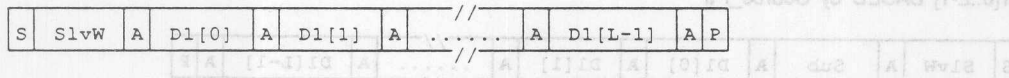
```

Description:

IIC_Write is the most basic procedure to write a message to a slave device.

I²C Protocol:

L = Count
D1[0..L-1] BASED by Source_Ptr



Example:

```

DECLARE Data_Buffer ( 4 ) BYTE ;
.....
CALL IIC_Write ( 0C2h, LENGTH ( Data_Buffer ), .Data_Buffer ) ;

```

PLM51 I²C software interface IIC51 (version 0.5) ETV/AN89004

3.1.4. IIC_Write_Sub.

Declaration:

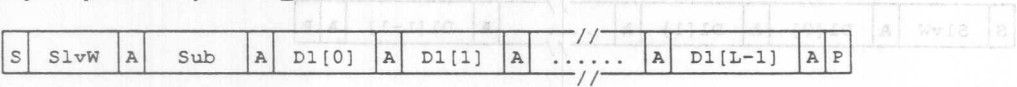
```
IIC_Write_Sub:
PROCEDURE ( Slave_Address, Count, Source_Ptr, Sub_Address )
[ BIT | BYTE ] EXTERNAL ;
DECLARE ( Slave_Address, Count, Source_Ptr, Sub_Address ) BYTE ;
END ;
```

Description:

IIC_Write_Sub writes a message preceded by a subaddress to a slave device.

I²C Protocol:

L = Count
Sub = Sub_Address
D1[0..L-1] BASED by Source_Ptr



Example:

```
DECLARE Data_Buffer ( 8 ) BYTE ;
.....
CALL IIC_Write_Sub ( 48h, LENGTH ( Data_Buffer ), .Data_Buffer, 2 ) ;
```

PLM51 I²C software interface IIC51 (version 0.5) ETV/AN89004

3.1.5. IIC_Write_Sub_SWInc.

Declaration:

```

IIC_Write_Sub_SWInc:
  PROCEDURE ( Slave_Address, Count, Source_Ptr, Sub_Address )
    DECLARE ( Slave_Address, Count, Source_Ptr, Sub_Address ) [ BIT | BYTE ] EXTERNAL;
  END ;

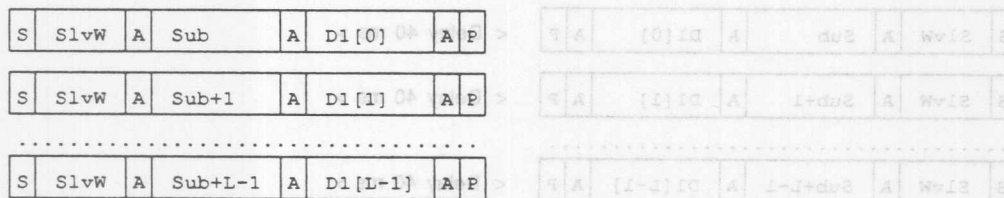
```

Description:

Some I²C devices addressed with a subaddress do not automatically increment the subaddress after reception of each byte. IIC_Write_Sub_SWInc can be used for such devices the same way IIC_Write_Sub is used. IIC_Write_Sub_SWInc splits up the message in smaller messages and increments the subaddress itself.

I²C Protocol:

L = Count
 Sub = Sub_Address
 D1[0..L-1] BASED by Source_Ptr



Example:

```

DECLARE Data_Buffer ( 6 ) BYTE ;
.....
CALL IIC_Write_Sub_SWInc ( 80h, LENGTH ( Data_Buffer ), .Data_Buffer, 0 ) ;

```

PLM51 I²C software interface IIC51 (version 0.5) ETV/AN89004

3.1.6. IIC_Write_Memory.

Declaration:

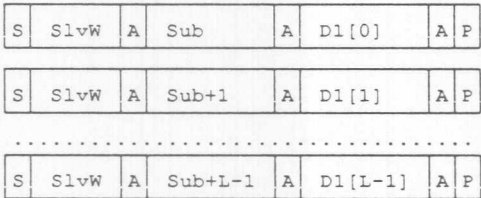
```
IIC_Write_Memory:
PROCEDURE ( Slave_Address, Count, Source_Ptr, Sub_Address )
[ BIT | BYTE ] EXTERNAL;
DECLARE ( Slave_Address, Count, Source_Ptr, Sub_Address ) BYTE ;
END ;
```

Description:

I²C Non-Volatile Memory devices (such as PCF8582) need an additional delay after writing a byte to it. IIC_Write_Memory can be used to write to such devices the same way IIC_Write_Sub is used. IIC_Write_Memory splits up the message in smaller messages and increments the subaddress itself. After transmission of each small message a delay of 40 milliseconds is inserted.

I²C Protocol:

L = Count
Sub = Sub_Address
D1[0..L-1] BASED by Source_Ptr



Example:

```
DECLARE Data_Buffer ( 10 ) BYTE ;
.....
CALL IIC_Memory ( 0A0h, LENGTH ( Data_Buffer ), Data_Buffer, 0F0h ) ;
```

PLM511²C software interface IIC51 (version 0.5) ETV/AN89004

3.1.7. IIC_Write_Sub_Write.

Declaration:

IIC_Write_Sub_Write:

PROCEDURE (Slave_Address, Count1, Source_Ptr1, Sub_Address, Count2, Source_Ptr2)

[BIT | BYTE] EXTERNAL ;

DECLARE (Slave_Address, Count1, Source_Ptr1, Sub_Address, Count2, Source_Ptr2)

BYTE ;

END ;

Description:

IIC_Write_Sub_Write writes 2 data blocks preceded by a subaddress in one message to a slave device. This procedure can be used for devices that need an extended addressing method, without the need to put all data into one large buffer. Such a device is the ECCT (I²C controlled teletext device; see example).

I²C Protocol:

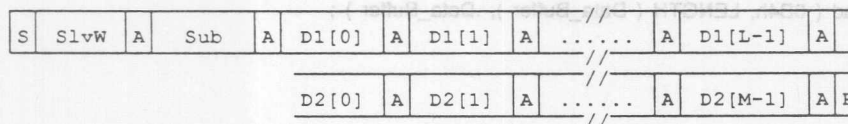
L = Count1

M = Count2

Sub = Sub_Address

D1[0..L-1] BASED by Source_Ptr1

D2[0..M-1] BASED by Source_Ptr2

Example:

PROCEDURE Write_CCT_Memory (Chapter, Row, Column, Data_Buf, Data_Count) ;

DECLARE (Chapter, Row, Column, Data_Buf, Data_Count) BYTE ;

/*

The extended address (CCT-Cursor) is formed by Chapter, Row and Column. These three bytes are written after the subaddress (8) followed by the actual data which will be stored relative to the extended address.

*/

CALL IIC_Write_Sub_Write (22h, 3, .Chapter, 8, Data_Buf, Data_Count) ;

END Write_CCT_Memory ;

PLM5112 software interface IIC51 (version 0.5) ETV/AN89004

3.1.8. IIC_Read.

Declaration:

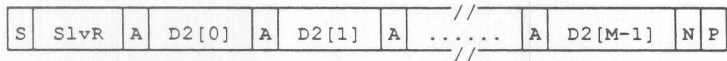
IIC_Read:
PROCEDURE (Slave_Address, Count, Dest_Ptr) [BIT | BYTE] EXTERNAL ;
DECLARE (Slave_Address, Count, Dest_Ptr) BYTE ;
END ;

Description:

IIC_Read is the most basic procedure to read a message from a slave device.

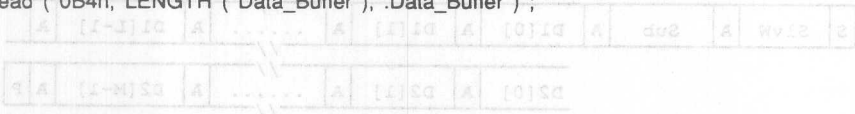
I²C Protocol:

M = Count
D2[0..M-1] BASED by Dest_Ptr



Example:

DECLARE Data_Buffer (4) BYTE ;
.....
CALL IIC_Read (0B4h, LENGTH (Data_Buffer), Data_Buffer) ;



Example:

PROCEDURE Write_CCT_Memory (Chapter, Row, Column, Data_Buf, Data_Count)
DECLARE (Chapter, Row, Column, Data_Buf, Data_Count) BYTE ;

The extended address (CCT-Cursor) is formed by Chapter, Row and Column. These three bytes are written after the subaddress (S) followed by the actual data which will be stored relative to the extended address.

CALL IIC_Write_Sub_Write (Sst, 3, Chapter, Row, Column, Data_Buf, Data_Count) ;

END Write_CCT_Memory ;

PLM51 I²C software interface IIC51 (version 0.5) ETV/AN89004

3.1.9. IIC_Read_Status.

Declaration:

```

IIC_Read_Status:
  PROCEDURE ( Slave_Address, Dest_Ptr ) [ BIT | BYTE | EXTERNAL ;
  DECLARE   ( Slave_Address, Dest_Ptr ) BYTE ;
  END ;

```

Description:

A lot of I²C devices have only a one status byte that can be read via I²C. IIC_Read_Status can be used for this purpose. IIC_Read_Status works the same as IIC_Read but the user does not have to pass a count parameter.

I²C Protocol:

M = Count
Status BASED by Dest_Ptr

S	SlvR	A	Status	N	P
---	------	---	--------	---	---

Example:

```

DECLARE Status_Byte BYTE ;
.....
CALL IIC_Read_Status ( 84h, .Status_Byte ) ;

```

PLM51 I²C software interface IIC51 (version 0.5) ETV/AN89004

3.1.10. IIC_Read_Sub.

Declaration:

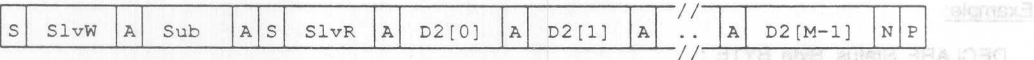
```
IIC_Read_Sub:
PROCEDURE ( Slave_Address, Count, Dest_Ptr, Sub_Address )
[ BIT | BYTE ] EXTERNAL ;
DECLARE ( Slave_Address, Count, Dest_Ptr, Sub_Address ) BYTE ;
END ;
```

Description:

IIC_Read_Sub reads a message from a slave device preceded by a write of the subaddress. Between writing the subaddress and reading the message an I²C restart condition is generated without surrendering the bus. This prevents other masters from accessing the slave device in between and overwriting the subaddress.

I²C Protocol:

M = Count
Sub = Sub_Address
D2[0..M-1]BASED by Dest_Ptr



Example:

```
DECLARE Data_Buffer ( 5 ) BYTE ;
.....
CALL IIC_Write_Sub ( 0A2h, LENGTH ( Data_Buffer ), .Data_Buffer, 2 ) ;
```

PLM51 I²C software interface IIC51 (version 0.5) ETV/AN89004

3.1.11. IIC_Write_Sub_Read.

Declaration:

IIC_Write_Sub_Read:

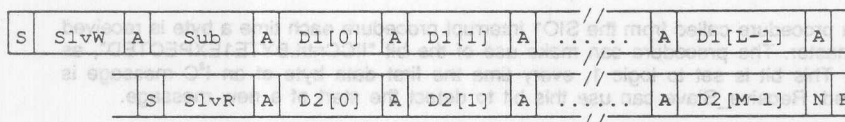
```
PROCEDURE ( Slave_Address, Count1, Source_Ptr1, Sub_Address, Count2, Dest_Ptr2 ) ;
[ BIT | BYTE | EXTERNAL ;
DECLARE ( Slave_Address, Count1, Source_Ptr1, Sub_Address, Count2, Dest_Ptr2 ) ;
BYTE ;
END ;
```

Description:

IIC_Write_Sub_Read writes a data block preceded by a subaddress, generates an I²C restart condition, and reads a data block. This procedure can be used for devices that need an extended addressing method. Such a device is the ECCT (I²C controlled teletext device; see example).

I²C Protocol:

```
L      = Count1
M      = Count2
Sub    = Sub_Address
D1[0..L-1] BASED by Source_Ptr1
D2[0..M-1] BASED by Dest_Ptr2
```



Example:

```
PROCEDURE Read_CCT_Memory ( Chapter, Row, Column, Data_Buf, Data_Count ) ;
DECLARE ( Chapter, Row, Column, Data_Buf, Data_Count ) BYTE ;
```

/*

The extended address (CCT-Cursor) is formed by Chapter, Row and Column. These three bytes are written after the subaddress (8). After that the actual data will be read relative to the extended address.

*/

```
CALL IIC_Write_Sub_Read ( 22h, 3, .Chapter, 8, Data_Buf, Data_Count ) ;
```

```
END Read_CCT_Memory ;
```

PLM51 I²C software interface IIC51 (version 0.5) ETV/AN89004

3.2. Slave Mode Functions.

I²C slave mode is provided by IIC51M only. All slave mode actions (except initialisation) take place in the SIO1 interrupt procedure. Slave mode I²C protocol is very application dependent. If a specific slave mode is required, the user has to modify three procedures in IIC51M at source level. The following sections describe these procedures. The program examples of the procedures implement an I²C slave protocol to read and write the microcontroller's on chip RAM via I²C. This can be a useful feature during program development and debugging.

3.2.1. Init_Slave.

This procedure is called from IIC_Init. In this procedure the user can initialise all static data concerning slave mode functions (if any).

Example:

```
Slave_Sub_Address:  db  1
...
Init_Slave:         mov  Slave_Sub_Address,#0    ; Initialise Sub Address
                   ret
```

3.2.2. Receive_Slave.

Receive_Slave is a procedure called from the SIO1 interrupt procedure each time a byte is received from another I²C master. The procedure can make use of the bit "IICNtrl.BYTE1EXPECTED", as defined in IIC51M. This bit is set to logic 1, every time the first data byte of an I²C message is about to be received. Receive_Slave can use this bit to detect the start of a new message.

Normally all bytes received from the other master will be acknowledged (i.e. SIO1 control bit Assert Acknowledge is set, AA = 1). If AA is cleared by Receive_Slave subsequent bytes in the message will be ignored and a negative acknowledge will be transmitted after reception of each byte. Note that the example does not make use of this feature.

Constraints:

- Receive_Slave must read the S1DAT register.
- Receive_Slave may clear the SIO1 control bit AA, to stop acknowledging data.
- Receive_Slave may not effect any other SIO1 hardware registers / bits.
- Receive_Slave is only allowed to use the accumulator and register R0 in the current registerbank.

Example:

```
Receive_Slave:  mov  a,S1DAT                ; Pick up data
               mov  r0,#Slave_Sub_Address  ; Prepare for 1st byte
               jbc  IICNtrl.BYTE1EXPECTED,Save_Byte ; Jump if 1st byte
               mov  r0,Slave_Sub_Address    ; Else data byte
               inc  Slave_Sub_Address        ; Postincrement Sub.
Save_Byte:     mov  @r0,a                   ; Save Data
               ret                          ; Exit
```


PLM51 I²C software interface IIC51 (version 0.5) ETV/AN89004

3.2.3. Send_Slave.

Send_Slave is a procedure called during the SIO1 interrupt procedure each time a byte has to be transmitted to another I²C master. This occurs after reception of I²C startcondition followed by the microcontroller's own slaveaddress (as passed to Init_IIC) with read-bit. Send_Slave will be called again after transmission of each subsequent byte, until a negative acknowledge is received from the reading I²C master.

Constraints:

- Send_Slave must write to the S1DAT register.
- Send_Slave may not effect any other SIO1 hardware registers / bits.
- Send_Slave is only allowed to use the accumulator and register R0 in the current registerbank.

Example:

```
Send_Slave:    mov r0,Slave_Sub_Address      ; Pick up Sub Address
               mov S1DAT,@r0                ; Send Data
               inc Slave_Sub_Address         ; Postincrement Sub.
               ret                           ; Exit
```

I²C driver routines for 8XC751/2 microcontrollers EIE/AN91007

Author: J.C.P.M. Pijnenburg, Eindhoven

1 Introduction

This report describes the I²C drivers which are written for the 8xC751/2. The report describes not only how to use the routines, but also the structure of the software. The software is written around a set of basic routines and a message handler. The message handler does not contain any specific 8xC751 code, so the software can be easily rewritten for any other bit level I²C interface by rewriting the set of basic routines. In the rest of this report when 8xC751 is written it means 8xC751/2

The package supports also the multimaster features of the I²C bus

The maximum bit rate possible when using those routines is approximately 70Kbit/sec.

References:

- The I²C-bus specification: 9398 358 10011
- 8051-based 8-bit Microcontrollers: Data Handbook IC20
- PLM51 I²C Software interface I2C51: ETV/AN89004

2 General

2.1 Memory Usage & file structure

The driver software consist of 3 main parts being:

- I²C message handler
- I²C basic routines
- I²C slave routines

During I²C usage it claims register bank 1, however register bank 1 does not contain any static I²C data and can be used by the application program outside the I²C routines (this data will be destroyed by I²C routines). The accumulator is also modified during I²C transfer.

The message handler uses a Message Control Block which consist of 8 bytes RAM. In those bytes, the following parameters are stored:

- for block 1 : I2C_ADDR_1, BUF_LEN and BUF_PTR_1
- for block 2 : I2C_ADDR_2, BUF_LEN and BUF_PTR_2

2 bytes of bit addressable RAM for STATUS and CONTROL information.

The STATUS byte is returned into the accumulator. If you do not need a detailed status, you can test the carry bit, this is a copy of the I2C_ERROR bit of the status register (returned in the acc.). The status register contains the following information:

bit:	name:	function:
0	RETRY_0	1
1	RETRY_1	1- Retry counter (0..7), as given during I2C_INIT
2	RETRY_2	1
3	I2C_ERR	I2C error if set (also available in carry)
4	TIME_ERR	Bus timeout occurred if set
5	RECOVER	- (no value for user) always 0
6	BUS_RECOVERED	If set, bus K recovered after timeout
7	NO_ACK	No acknowledge received

I²C driver routines for 8XC751/2 microcontrollers EIE/AN91007

The slave function uses 2 bytes of ram, those contain the own slave address (OWN_SLV_ADDR) and a pointer the slave transmit/receive buffer of the 8xC751. This is the buffer from/in which the 8xC751 gets/stores the data bytes in slave mode.

The I²C module is built around a message handler which calls basic functions such as I2C_TRX_BYTE and I2C_START. Each function calls the message handler after loading the correct mask into the I2C_CTRL byte.

Filename :	Function :	Include/Link:	Code size (byte):
I2C_DATA.GLO	I ² C global data definitions	I, each I ² C function and assembler main	0
I2C_DATA.LOC	I ² C local data definitions	I, each I ² C function	0
I2C_CODE.GLO	I ² C global function definitions	I, assembler main	0
I2C_INIT.ASM	Init_I2C (does not use message handler)	Link	22
I2C_DEF.ASM	Define MCB & _I2C_xxx_BYTES	Link	0
I2C_HAND.ASM	I ² C Message handler	Link	144
I2C_BASI.ASM	I ² C basic functions, and TI interrupt handling	Link	293
I2C_TDEV.ASM	I ² C Test_Device	Link, if used	5
I2C_WRIT.ASM	I ² C Write	Link, if used	5
I2C_WSUB.ASM	I ² C Write_Sub	Link, if used	5
I2C_WSWI.ASM	I ² C Write_Sub_SWinc & Write_Mem	Link, if used	36
I2C_WSUW.ASM	I ² C Write_Sub_Write	Link, if used	5
I2C_WSUR.ASM	I ² C Write_Sub_Read	Link, if used	5
I2C_WCOW.ASM	I ² C Write_Com_Write	Link, if used	11
I2C_WREW.ASM	I ² C Write_Rep_Write	Link, if used	5
I2C_WRER.ASM	I ² C Write_Rep_Read	Link, if used	5
I2C_READ.ASM	I ² C Read and Read_Status	Link, if used	11
I2C_RSUB.ASM	I ² C Read_Sub	Link, if used	5
I2C_RRER.ASM	I ² C Read_Rep_Read	Link, if used	5
I2C_RREW.ASM	I ² C Read_Rep_Write	Link, if used	5
I2C_SLAV.ASM	I ² C Slave Function	Link	56

The total memory usage for the full package is

ROM	: 520 (single function) to 623 (all functions) bytes
RAM	byte addressable : 8 bytes
	bit addressable : 2 bytes
	register bank 1 : 8 bytes

The message handler, causes the other functions to be very small, to further reduce the code, all functions are placed in separate modules, which are put into a library I2C_751.LIB. If this library is linked to an application program, only the object modules which are used by the application program are linked in the output file.

The I2C_CODE.H file contains the references to the separate functions (EXTRN CODE definitions). The use must not include this file into main, but only copy the definitions which he needs into the source file. If this file is included, all functions will be linked, the library approach is of no use in this case.

I²C driver routines for 8XC751/2 microcontrollers EIE/AN91007

2.2 Retries

During initialisation the user defines whether he wants to use retries or not. If an I²C message fails, and retries ≥ 0 , the program restarts the message. This is done for at most 7 times. If the message remains unsuccessful, the message handler returns to the main program, indicating that the message has failed (carry set).

2.3 Error Handling

In case of an error while operating as master, the program returns to the message handler, the message handler. The message handler decides whether to invoke a retry or to return to the main program. The I²C interface of the 8xC751 generates a timeout interrupt if the bus hangs for more than 1022 cycles, in this case, if the 8xC751 is master (RECOVER = 1), a bus recover routine is started, if the 8xC751 is not master the I²C bus is released. Retries are only invoked in the master situation.

2.4 Development tools

The following software tools from Tasking/BSO are used for program development:

- OM4142 Cross Assembler 8051 for DOS: V3.0b
- OM4144 PL/M 8051 Compiler for DOS: V3.0a
- OM4136 C8051 Compiler for DOS: V1.1a
- OM4129 XRAY51 debugger: V1.4c

I²C driver routines for 8XC751/2 microcontrollers EIE/AN91007

3 Master routines

3.1 Message Handler

To make the I²C protocols as described in paragraphs 3.2 to 3.17, an I²C message handler is written. The message control block (I2C_MCB) together with the I²C control byte (I2C_CTRL) form the input for the message handler. The I2C_MCB includes 6 bytes of data, containing:

- I2C_ADDR1, first address in the protocol
- BUF_LEN_1, length of the first data buffer
- BUF_PTR_1, pointer to the first data buffer
- I2C_ADDR2, second address in the protocol
- BUF_LEN_2, length of the second data buffer
- BUF_PTR_2, pointer to the second data buffer

The I2C_CTRL byte is bitaddressable it contains 8 bits who determine the flow through the message handler. This byte must be loaded with the corresponding mask before starting the message handler.

The I2C_CTRL byte contains the following bits:

- REP_STRT_BLK1 must we send a repeated start before the first data block ? (0=NO,1=YES)
- RWN_BLK1 read (1) or write (0) the first block of data
- ADDR2 is there a second address in the protocol ? (0=NO,1=YES)
- ADDR2_SUB is the 2nd address a sub address, only relevant if ADDR2=1.
- BLOCK2 is there a second block of data in the protocol ? (0=NO,1=YES)
- RWN_BLK1 read (1) or write (0) the first block of data
- REP_STRT_BLK1 must we send a repeated start before the second data block ?
- TEST_DEVICE is it the test device protocol (paragraph 4.4)

When a I²C protocol is handled successful by the message handler, it returns control to the main program, if not it can do a retry by resending the message (maximal 5 retries are possible).

The message handler return value is stored in the I2C_STAT byte, The I2C_ERROR bit indicates wether the transfer has succeeded (Note: better is to copy this byte into Acc before returning the control to the main program, this way a byte can be saved (I2C_STAT can be placed in register bank one) and the main program can do a JZ/JNZ test (must be changed))

I²C driver routines for 8XC751/2 microcontrollers EIE/AN91007

3.2 I2C_INIT

Description:

Init_I2C must be called after RESET, before any procedure is called. The I²C interface and I²C interrupt will be enabled (SETB ETI, EI2 and EA). Own_Slave_Address is passed to Init_I2C for use as slave. Slave_Sub_Address is the pointer to a DATA buffer that is used for data transfer in slave mode. When used as master in a single master system, these parameters are not used. Retry is the number of retries on messages when an error occurs. 0 means no retry (just 1 attempt to send a message), while the maximum amount of retries is 7.

I²C Protocol:

none (no action at I²C bus)

Calling Sequence:

```
C      : I2C_INIT(Own_Slv_Addr,Slv_Buf_Addr,Retry);
PL/M51 : I2C_INIT(Own_Slv_Addr,Slv_Buf_Addr,Retry);
Assembler : %I2C_INIT(Own_Slv_Addr,Slv_Buf_Addr,Retry);
           (macro call)
```

Parameters:

Own_Slave_Adr : 8xC751 own slave address
 Slave_Buffer_Adr : Base address of buffer, to transmit data from, or receive data in, when 8xC751 is in slave mode.
 Retry : Number of times to do a retry in case of an error. 0 = No Retry, maximum retries is 7.

NOTE:

The Init_I2C function enables the I²C watchdog timer interrupt (TI). This watchdog generates an interrupt when during an I²C transfer, SCL is hold longer than 1022 machine cycles (ca. 760 μ s at 16 MHz). If this time is too short for your application, you can disable the TI (CLR ETI). In this case, the main program must check if a bus hangup occurs, and take proper action when the bus is hangup.

I²C driver routines for 8XC751/2 microcontrollers EIE/AN91007

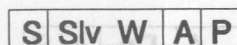
3.3 I2C_TEST_DEVICE

Description:

I2C_Test_Device just sends the slave address to the PC bus. It can be used to check the presence of a device on the PC bus

I²C Protocol:

Slv_W : Slave_Adr + Write bit



Device is present



Device is not present

Calling Sequence:

C : I2C_TEST_DEVICE(Slv_Adr);
 PL/M51 : I2C_TEST_DEVICE(Slv_Adr);
 Assembler : %I2C_TEST_DEVICE(Slv_Adr);
 (macro call)

Parameters:

Slave_Adr : Slave address of the device to be tested.

3.4 I2C_WRITE

Description:

I2C_Write is the most basic procedure to write a message to a slave device.

I2C Protocol:

Slv_W : Slave_Adr + Write bit
D0..Dn : Data bytes



Calling Sequence:

C : I2C_WRITE(Slv_Adr,Count,Source_Ptr);
PL/M51 : I2C_WRITE(Slv_Adr,Count,Source_Ptr);
Assembler : %I2C_WRITE(Slv_Adr,Count,Source_Ptr);

Parameters:

Slave_Adr : Slave address of the device to write to.
Count : Number of bytes to transmit (D0 .. Dn, n= count-1)
Source_Ptr : Pointer to data buffer, to transmit bytes from.
(macro call)

I²C driver routines for 8XC751/2 microcontrollers EIE/AN91007

3.5 I2C_WRITE_SUB

Description:

I2C_Write_Sub writes a message preceded by a sub-address to a slave device

PC Protocol:

- Slv_W : Slave_Adr + Write bit
- Sub : Sub_Adr
- D0..Dn : Data bytes



Calling Sequence:

- C : I2C_WRITE_SUB(Slv_Adr,Count,Source_Ptr,Sub_Adr);
- PL/M51 : I2C_WRITE_SUB(Slv_Adr,Count,Source_Ptr,Sub_Adr);
- Assembler : %I2C_WRITE_SUB(Slv_Adr,Count,Source_Ptr,Sub_Adr);
(macro call)

Parameters:

- Slave_Adr : Slave address of the device to write to.
- Count : Number of bytes to transmit (D0 .. Dn, n= count-1)
- Source_Ptr : Pointer to data buffer, to transmit bytes from.
- Sub_Adr : Sub address.

I²C driver routines for 8XC751/2 microcontrollers EIE/AN91007

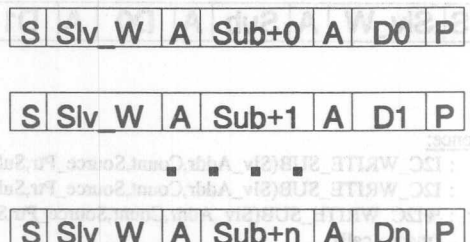
3.6 I2C_WRITE_SUB_SWINC

Description:

Some I²C devices addressed with a sub-address do not automatically increment the sub-address after reception of each byte. I2C_Write_Sub_SWInc can be used for such devices the same way as I2C_Write_Sub is used. I2C_Write_Sub_SWInc splits up the message in smaller messages and increments the sub-address itself.

I²C Protocol:

Slv_W : Slave_Adr + Write bit
 Sub+x : Sub_Adr+x
 D0..Dn : Data bytes

Calling Sequence:

C : I2C_WRITE_SUB_SWINC(Slv_Adr,Count,Source_Ptr,Sub_Adr);
 PL/M51 : I2C_WRITE_SUB_SWINC(Slv_Adr,Count,Source_Ptr,Sub_Adr);
 Assembler : %I2C_WRITE_SUB_SWINC(Slv_Adr,Count,Source_Ptr,Sub_Adr);
 (macro call)

Parameters:

Slave_Adr : Slave address of the device to write to.
 Count : Number of bytes to transmit (D0 .. Dn, n= count-1)
 Source_Ptr : Pointer to data buffer, to transmit bytes from.
 Sub_Adr : Sub address.

I²C driver routines for 8XC751/2 microcontrollers EIE/AN91007

3.7 I2C_WRITE_MEMORY

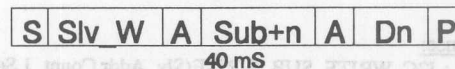
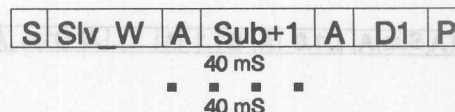
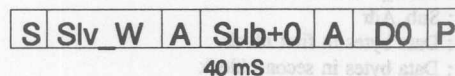
Description:

I²C Non-Volatile Memory devices (such as PCF8582) need an additional delay after writing a byte to it. I2C_Write_Memory can be used to write to such devices the same way I2C_Write_Sub is used.

I2C_Write_Memory splits up the message in smaller messages and increments the sub-address itself. After transmission of each message a delay of 40 milliseconds ($f_{\text{xtal}} = 16\text{MHz}$) is inserted.

PC Protocol:

Slv_W : Slave_Adr + Write bit
Sub+x : Sub_Adr+x
D0..Dn : Data bytes

Calling Sequence:

C : I2C_WRITE_MEMORY(Slv_Adr,Count,Source_Ptr,Sub_Adr);
PL/M51 : I2C_WRITE_MEMORY(Slv_Adr,Count,Source_Ptr,Sub_Adr);
Assembler : %I2C_WRITE_MEMORY(Slv_Adr,Count,Source_Ptr,Sub_Adr);
(macro call)

Parameters:

Slave_Adr : Slave address of the device to write to.
Count : Number of bytes to transmit (D0 .. Dn, n= count-1)
Source_Ptr : Pointer to data buffer, to transmit bytes from.
Sub_Adr : Sub address.

I²C driver routines for 8XC751/2 microcontrollers EIE/AN91007

3.8 I2C_WRITE_SUB_WRITE

Description:

I2C_Write_Sub_Write writes 2 data blocks preceded by a sub-address in one message to a slave device. This procedure can be used for devices that need an extended addressing method, without the need to put all data into one large buffer. Such a device is the ECCT (I²C controlled teletext device; see example).

I²C Protocol:

Slv_W : Slave_Adr + Write bit
 Sub : Sub_Adr
 D1.0..D1.n : Data bytes in first block
 D2.0..D2.p : Data bytes in second block

S	Slv_W	A	Sub	A	D1.0	A	D1.1	A	A	D1.n	A	D2.0	A	A	D2.p	A	P
---	-------	---	-----	---	------	---	------	---	-------	---	------	---	------	---	-------	---	------	---	---

Calling Sequence:

```
C      : I2C_WRITE_SUB_WRITE(Slv_Adr,Count_1,Source_Ptr_1,
                               Sub_Adr,Count_2,Source_Ptr_2);
PL/M51 : I2C_WRITE_SUB_WRITE(Slv_Adr,Count_1,Source_Ptr_1,
                               Sub_Adr,Count_2,Source_Ptr_2);
Assembler : %I2C_WRITE_SUB_WRITE(Slv_Adr,Count_1,Source_Ptr_1,
                                   Sub_Adr,Count_2,Source_Ptr_2);
          (macro call)
```

Parameters:

Slave_Adr_1 :Slave address of the device to write to.
 Count_1 :Number of bytes to transmit in first block (D1.0 .. D1.n, n= count_1-1)
 Source_Ptr_1 :Pointer to first block of data, to transmit.
 Sub_Adr :Sub address.
 Count_2 :Number of bytes to transmit in second block (D2.0 .. D2.p, p= count_2-1)
 Source_Ptr_2 :Pointer to second block of data to transmit.

I²C driver routines for 8XC751/2 microcontrollers EIE/AN91007

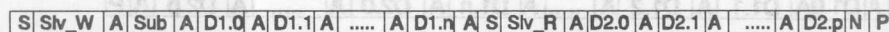
3.9 I2C_WRITE_SUB_READ

Description:

I2C_Write_Sub_Read writes a data block preceded by a sub-address, generates an I²C restart condition, and reads a data block. This procedure can be used for devices that need an extended addressing method. Such a device is the ECCT.

I²C Protocol:

Slv_W : Slave_Adr + Write bit
 Slv_R : Slave_Adr + Read bit
 Sub : Sub_Adr
 D1.0..D1.n : Data bytes in first block (write)
 D2.0..D2.p : Data bytes in second block (read)

Calling Sequence:

C : I2C_WRITE_SUB_READ(Slv_Adr,Count_1,Source_Ptr,Sub_Adr,Count,Dest_Ptr);
 PL/M51 : I2C_WRITE_SUB_READ(Slv_Adr,Count_1,Source_Ptr,Sub_Adr,Count,Dest_Ptr);
 Assembler : %I2C_WRITE_SUB_READ(Slv_Adr,Count_1,Source_Ptr,Sub_Adr,Count,Dest_Ptr);
 (macro call)

Parameters:

Slave_Adr_1 : Slave address of the device to write and read to/from.
 Count_1 : Number of bytes to transmit (D1.0 .. D1.n, n= count-1)
 Source_Ptr_1 : Pointer to first block of data to transmit.
 Sub_Adr : Sub address.
 Count_2 : Number of bytes to transmit in second block (D2.0 .. D2.p, p= count_2-1)
 Dest_Ptr_2 : Pointer buffer to receive second block of data in.

I²C driver routines for 8XC751/2 microcontrollers EIE/AN91007

3.10 I2C_WRITE_COM_WRITE

Description:

I2C_Write_Com_Write writes two data blocks from different data buffers in one message to a slave receiver. This procedure can be used for devices where the message consists of 2 different data blocks. Such devices are for instance LCD-drivers, where the first part of the message consists of addressing and control information, and the second part is the data string to be displayed.

I²C Protocol:

- Slv_W : Slave_Adr + Write bit
- D1.0..D1.n : Data bytes in first block (write)
- D2.0..D2.p : Data bytes in second block (write)



Calling Sequence:

- C : I2C_WRITE_COM_WRITE(Slv_Adr,Count_1,Source_Ptr_1,Count_2,Source_Ptr_2);
- PL/M51 : I2C_WRITE_COM_WRITE(Slv_Adr,Count_1,Source_Ptr_1,Count_2,Source_Ptr_2);
- Assembler : %I2C_WRITE_COM_WRITE(Slv_Adr,Count_1,Source_Ptr_1,Count_2,Source_Ptr_2);
(macro call)

Parameters:

- Slave_Adr :Slave address of the device to write to.
- Count_1 :Number of bytes to transmit in first block (D1.0 .. D1.n, n= count_1-1)
- Source_Ptr_1 :Pointer to first block of data, to transmit.
- Count_2 :Number of bytes to transmit in second block (D2.0 .. D2.p, p= count_2-1)
- Source_Ptr_2 :Pointer to second block of data to transmit.

I²C driver routines for 8XC751/2 microcontrollers EIE/AN91007

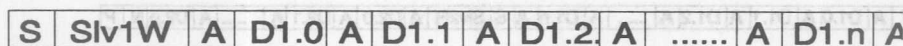
3.11 I2C_WRITE_REP_WRITE

Description:

Two data strings are sent to separate slave devices, separated with a repeat START condition. This has the advantage that the bus does not have to be released with a STOP condition before the transfer from the second slave.

I²C Protocol:

Slv1W : Slave_Adr_1 + Write bit
Slv2W : Slave_Adr_2 + Write bit
D1.0..D1.n : Data bytes in first block (write to first slave)
D2.0..D2.p : Data bytes in second block (write to second slave)



Calling Sequence:

```

C          : I2C_WRITE_REP_WRITE(Slv_Addr,Count_1,Source_Ptr_1,
                                     Sub_Addr,Count_2,Source_Ptr_2);
PL/M51    : I2C_WRITE_REP_WRITE(Slv_Addr,Count_1,Source_Ptr_1,
                                     Sub_Addr,Count_2,Source_Ptr_2);
Assembler : %I2C_WRITE_REP_WRITE(Slv_Addr,Count_1,Source_Ptr_1,
                                     Sub_Addr,Count_2,Source_Ptr_2);
          (macro call)

```

Parameters:

Slave_Adr_1	:Slave address of first device to write to.
Count_1	:Number of bytes to transmit in first block (D1.0 .. D1.n, n= count_1-1)
Source_Ptr_1	:Pointer to first block of data, to transmit.
Slave_Adr_2	:Slave address of second device to write to.
Count_2	:Number of bytes to transmit in second block (D2.0 .. D2.p, p= count_2-1)
Source_Ptr_2	:Pointer to second block of data to transmit.

I²C driver routines for 8XC751/2 microcontrollers EIE/AN91007

3.12 I2C_WRITE_REP_READ

Description:

A data string is sent and received to/from two separate slave devices, separated with a repeat START condition. This has the advantage that the bus does not have to be released with a STOP condition before the transfer from the second slave.

PC Protocol:

Slv1W : Slave_Adr_1 + Write bit
 Slv2R : Slave_Adr_2 + Read bit
 D1.0..D1.n : Data bytes in first block (write to first slave)
 D2.0..D2.p : Data bytes in second block (write to second slave)

S	Slv1W	A	D1.0	A	D1.1	A	D1.2	A	A	D1.n	A	S	Slv2R	A	D2.0	A	D2.1	A	A	D2.p	N	P
---	-------	---	------	---	------	---	------	---	-------	---	------	---	---	-------	---	------	---	------	---	-------	---	------	---	---

Calling Sequence:

C : I2C_WRITE_REP_READ(Slv_Adr,Count_1,Source_Ptr,Sub_Adr,Count_2,Dest_Ptr);
 PL/M51 : I2C_WRITE_REP_READ(Slv_Adr,Count_1,Source_Ptr,Sub_Adr,Count_2,Dest_Ptr);
 Assembler : %I2C_WRITE_REP_READ(Slv_Adr,Count_1,Source_Ptr,Sub_Adr,Count_2,Dest_Ptr);
 (macro call)

Parameters:

Slave_Adr_1 : Slave address of first device to write to.
 Count_1 : Number of bytes to transmit in first block (D1.0 .. D1.n, n= count_1-1)
 Source_Ptr_1 : Pointer to first block of data, to transmit.
 Slave_Adr_2 : Slave address of second device to read from.
 Count_2 : Number of bytes to transmit in second block (D2.0 .. D2.p, p= count_2-1)
 Dest_Ptr_2 : Pointer buffer to receive second block of data in.

I²C driver routines for 8XC751/2 microcontrollers

EIE/AN91007

3.13 I2C_READ

Description:
I2C_Read is the most basic procedure to read a message from a slave device.

I²C Protocol:
Slv_R : Slave_Adr + Read bit
D0..Dn : Data bytes



Calling Sequence:
C : I2C_READ(Slv_Adr,Count,Dest_Ptr);
PL/M51 : I2C_READ(Slv_Adr,Count,Dest_Ptr);
Assembler : %I2C_READ(Slv_Adr,Count,Dest_Ptr);
(macro call)

Parameters:
Slave_Adr : Slave address of the device to be tested.
Count : Number of bytes to transmit (D0 .. Dn, n= count-1)
Dest_Ptr : Pointer to data buffer, to receive bytes in.

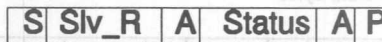
3.14 I2C_READ_STATUS

Description:

Several I2C devices can send a one byte status-word via the bus. I2C_Read_Status can be used for this purpose. I2C_Read_Status works the same way as I2C_Read but the user does not have to pass a count parameter.

I2C Protocol:

Slv_R : Slave_Adr + Read bit
Status : Status byte



Calling Sequence:

C : I2C_READ_STATUS(Slv_Adr, Dest_Ptr);
PL/M51 : I2C_READ_STATUS(Slv_Adr, Dest_Ptr);
Assembler : %I2C_READ_STATUS(Slv_Adr, Dest_Ptr);
(macro call)

Parameters:

Slave_Adr : Slave address of the device to be tested.
Count : Number of bytes to transmit (D0 .. Dn, n= count-1)
Dest_Ptr : Pointer to data buffer, to receive status byte in.

I²C driver routines for 8XC751/2 microcontrollers

3.15 I2C_READ_SUB

Description:
I2C_Read_Sub reads a message from a slave device, preceded by a write of the sub-address. Between writing the sub-address and reading the message an I²C restart condition is generated without releasing the bus. This prevents other masters from accessing the slave device in between and overwriting the sub-address.

PC Protocol:

- Slv_W : Slave_Adr + Write bit
- Slv_R : Slave_Adr + Read bit
- Sub : Sub_Adr



Calling Sequence:

- C : I2C_READ_SUB(Slv_Adr,Count,Dest_Ptr,Sub_Adr);
 - PL/M51 : I2C_READ_SUB(Slv_Adr,Count,Dest_Ptr,Sub_Adr);
 - Assembler : %I2C_READ_SUB(Slv_Adr,Count,Dest_Ptr,Sub_Adr);
- (macro call)

Parameters:

- Slave_Adr : Slave address of the device to be tested.
- Count : Number of bytes to transmit (D0 .. Dn, n= count-1)
- Dest_Ptr : Pointer buffer to receive bytes in.
- Sub_Adr : Sub address.

I²C driver routines for 8XC751/2 microcontrollers EIE/AN91007

3.16 I2C_READ_REP_READ

Description:

Two data strings are read from separate slave devices, separated with a repeat START condition. This has the advantage that the bus does not have to be released with a STOP condition before the transfer from the second slave.

PC Protocol:

Slv1R : Slave_Adr_1 + Read bit
 Slv2R : Slave_Adr_2 + Read bit
 D1.0..D1.n : Data bytes in first block (read from first slave)
 D2.0..D2.p : Data bytes in second block (read from second slave)

S	Slv_R	A	D1.0	A	D1.1	A	D1.2	A	A	D1.n	N	S	Slv_R	A	D2.0	A	D2.1	A	A	D2.p	N	P
---	-------	---	------	---	------	---	------	---	-------	---	------	---	---	-------	---	------	---	------	---	-------	---	------	---	---

Calling Sequence:

C : I2C_READ_REP_READ(Slv_Adr,Count_1,Dest_Ptr_1,Sub_Adr,Count_2,Dest_Ptr_2);
 PL/M51 : I2C_READ_REP_READ(Slv_Adr,Count_1,Dest_Ptr_1,Sub_Adr,Count_2,Dest_Ptr_2);
 Assembler : %I2C_READ_REP_READ(Slv_Adr,Count_1,Dest_Ptr_1,Sub_Adr,Count_2,Dest_Ptr_2);
 (macro call)

Parameters:

Slave_Adr_1 : Slave address of first device to write to.
 Count_1 : Number of bytes to transmit in first block (D1.0 .. D1.n, n= count_1-1)
 Dest_Ptr_1 : Pointer buffer to receive first block of data in.
 Slave_Adr_2 : Slave address of second device to read from.
 Count_2 : Number of bytes to transmit in second block (D2.0 .. D2.p, p= count_2-1)
 Dest_Ptr_2 : Pointer buffer to receive second block of data in.

I²C driver routines for 8XC751/2 microcontrollers EIE/AN91007

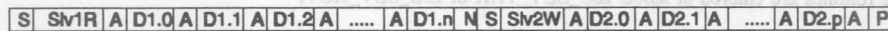
3.17 I2C_READ_REP_WRITE

Description:

A data string is received and send from/to two separate slave devices, separated with a repeat START condition. This has the advantage that the bus does not have to be released with a STOP condition before the transfer from the second slave.

I²C Protocol:

Slv1R : Slave_Adr_1 + Read bit
 Slv2W : Slave_Adr_2 + Write bit
 D1.0..D1.n : Data bytes in first block (read from first slave)
 D2.0..D2.p : Data bytes in second block (read from second slave)



Calling Sequence:

C : I2C_READ_REP_WRITE(Slv_Adr,Count_1,Dest_Ptr_1,Sub_Adr,Count_2,Source_Ptr);
 PL/M51 : I2C_READ_REP_WRITE(Slv_Adr,Count_1,Dest_Ptr_1,Sub_Adr,Count_2,Source_Ptr);
 Assembler : %I2C_READ_REP_WRITE(Slv_Adr,Count_1,Dest_Ptr_1,
 Sub_Adr,Count_2,Source_Ptr);

(macro call)

Parameters:

Slave_Adr_1 : Slave address of first device to write to.
 Count_1 : Number of bytes to transmit in first block (D1.0 .. D1.n, n= count_1-1)
 Dest_Ptr_1 : Pointer buffer to receive first block of data in.
 Slave_Adr_2 : Slave address of second device to read from.
 Count_2 : Number of bytes to transmit in second block (D2.0 .. D2.p, p= count_2-1)
 Source_Ptr_2 : Pointer buffer to transmit second block of data from.

4 Slave routines

The slave-mode protocol is very application dependent. In this note the basic slave-receive and slave-transmit routines are given and should be considered as examples. The user may for instance send NO_ACK after receiving a number of bytes to signal to the master-transmitter that a data buffer is full. A listing of the slave routines is given in appendix III

The I²C slave function has two entries:

1. **The I²C interrupt**, this can only occur at an idle slave, because when a transmission is in progress the I²C interrupt is disabled.
2. **Through the master routines**, during transmission of a slave-address in master-mode, arbitration is lost to another master. The interface must then switch to slave-receiver mode to check if this other master wants to address the 8xC751 I²C interface. If the 8xC751 recognises his own slave address, the slave mode routines are entered at labels I2C_SLV_TRX or I2C_SLV_RCV.

Interfacing the master routines, if the user wants to adapt the slave routines to his own needs, he has to keep in mind that the master routines use the I2C_SLV_TRX and I2C_SLV_RCV entries. The I²C slave routines are entered after the acknowledge has been send, therefor the ATN flag will be set when entering the slave routines at I2C_SLV_TRX or I2C_SLV_RCV.

The slave routines as given, make use of a single data buffer. When addressed as slave transmitter, data bytes from the data buffer are transmitted over the I²C bus until a not acknowledge or stop is received. When addressed as slave receiver, the data form the I²C bus is received into the data buffer until a not acknowledge or a stop is received.

The data buffer is initialised during the Init_I2C function, one of the parameters of this function is the pointer to the data buffer (SLV_BUF_PTR DS 1).

4.1 Slave Transmitter

The slave transmitter function transmits data bytes from the 8x751 data buffer (ACALL I2C_TRX_BYTE) until a not acknowledge or a stop is received. The function is also exit on an I²C error. The function is exit with the ATN bit set.

4.2 Slave Receiver

The slave receiver function receivers data bytes into the 8x751 data buffer (ACALL I2C_RCV_BYTE) until a stop is received. The function is also exit on an I²C error. The function is exit with the ATN bit set. If a byte has been received, an acknowledge is sent.

I²C driver routines for 8XC751/2 microcontrollers EIE/AN91007

5 Examples

5.1 Introduction

Some examples are given how to use the I²C routines in an application program. Examples are given for an assembly, PL/M and C program. The program displays time from the PCF8583P clock/calendar/RAM on an LCD display driven by the PCF8577. The example can be executed on the OM4151 I²C evaluation board.

5.2 Using the routines in assembly sources

Appendix VII shows the listing of the example program. The most important aspect when using the I²C routines, is preparing the input parameters before the sub-routine call. The parameters must be transferred to the MCB (Message Control Block). Below are 2 examples of how to transfer the necessary parameters to MCB (I²C_Read and I²C_Write_Sub_Read)

```
MOV _I2C_MCB,#Slave_Adr
MOV _I2C_MCB+1,#Count_1
MOV _I2C_MCB+2,#Dest_Ptr_1
ACALL _I2C_READ
```

```
MOV _I2C_MCB,#Sl_Adr
MOV _I2C_MCB+1,#Cnt_1
MOV _I2C_MCB+2,#S_Ptr_1
MOV _I2C_MCB+3,#Sub_Adr
MOV _I2C_MCB+4,#Cnt_2
MOV _I2C_MCB+5,#S_Ptr_2
ACALL _I2C_WRITE_SUB_READ
```

Note that the order of defining the parameters is the same as in PL/M- and C-calls (Calling sequences in paragraphs 3.2 to 3.17). An easier way to call the routines is to make a macro that includes the to transfer of the parameters.

The example program makes use of macros. I²C_Read is then called in the following way:

```
%I2C_READ(Slave_Adr,Count_1,Source_Ptr_1);
```

Note that in the listing the macro call is replaced by the contents of the macro.

The macro must be written as follows:

```
%* DEFINE (I2C_READ(Slave_Adr,Count_1,Dest_Ptr_1))
(
  MOV _I2C_MCB,%%Slave_Adr
  MOV _I2C_MCB+1,%%Count_1
  MOV _I2C_MCB+2,%%Dest_Ptr_1
  ACALL _I2C_READ
)
```

I²C driver routines for 8XC751/2 microcontrollers EIE/AN91007

File I2C_MAC.DEF contains the macro calls for the routines as described in paragraphs 3.2 to 3.17. This file should be included in all assembler modules in which calls to the I²C routines are made.

The file I2C_CODE.GLO contains the global function definitions (EXTRN CODE) of the I²C functions, copy the ones you need into your application. The file I2C_DATA.GLO contains the global data definitions of the I²C functions. Therefore this files must also be included in all assembler modules in which calls to the I²C routines are made.

All I²C routines return a status into the CY-bit. If the CY-bit is set, an error has occurred.

5.3 Using the routines in PL/M-51 sources

Appendix VIII shows the listing of the example program in PL/M-51. All procedures return a BIT value. The file I2C_PL/M.H contains the procedure declarations, this file can be included in the modules which call I²C routines. The routines are used the same way as in the examples of paragraph 5.2

5.4 Using the routines in C sources

Appendix IX shows the listing of the example program in C. All functions are return a bit value. The file I2C_C.H contains the function prototypes, this file can be included in the modules which call I²C routines. The routines are used the same way as in the examples of paragraph 5.2

I²C driver routines for 8XC751/2 microcontrollers EIE/AN91007

```

*=====*
*
* PACKAGE: I2C drivers for 8xC751/2 microcontroller
*
* DESCRIPTION: To use the package just link the library: I2c_751.lib
*              to your application program
*
* NOTES: If you use the package with assembler sources, you must include
*         \USER\INCLUDE\I2C_DATA.GLO and \USER\INCLUDE\I2C_MAC.DEF into
*         your main application(s).
*         \USER\INCLUDE\I2C_CODE.GLO contains external code definitions,
*         select the ones you need and copy them into your main application*
*         If you include this file, the linker assumes that you use all I2C*
*         functions and therefor links the complete package to your
*         application (in this case the library approach is of no use!)
*
*         If you use the package with PLM sources, you must include
*         \USER\INCLUDE\I2C_PLM.H in each file which uses an I2C function
*
*         If you use the package with C sources, you must include
*         \USER\INCLUDE\I2C_PLM.C in each file which uses an I2C function
*
*=====*

```

CONTENTS OF DISK

The disk contains 3 directories:

1:\USER :This directory contains 2 directories:

```

\INCLUDE :
    I2C_PLM.H      :PLM header file
    I2C_C.H        :C header file
    I2C_MAC.DEF    :ASM header file,
                   Macro definitions for ASM function calls
    I2C_DATA.GLO   :I2C global data (assembler only)
    I2C_DATA.LOC   :I2C local data (assembler only, not for user)
    I2C_CODE.GLO   :I2C extern code definitions (assembler only)
    REG751.H       :8xC751 register file

\LIB :
    LIB.BAT        :example batch file to create library
    I2C_751.LIB     :8xC751/2 I2C driver library

```

2:\EXAMPLE :This directory contains 3 directories

```

\DEMO_ASM      :Assembly example
\DEMO_PLM      :PL/M example
\DEMO_C        :C example

```

3:\SOURCE :This directory contains the source files of the modules that are
put in library with I2C_751.LIB

I²C driver routines for 8XC751/2 microcontrollers EIE/AN91007

I²C Master routines

¹²C driver routines for 8XC751/2 microcontrollers

```

$ TITLE (I2C_DEF.ASM)
;
;*
;* INCLUDE FILE: I2C_DEF.ASM
;* PACKAGE : I2C
;*
;-----
;* This file must be LINKED to each I2C sub function
;*
;-----
USING(1)

;-----
;* GLOBAL DATA DECLARATIONS
;*
PUBLIC      I2C_MCB
PUBLIC      I2C_CTRL
PUBLIC      I2C_STAT
PUBLIC      OWN_SLV_ADDR
PUBLIC      SLV_BUF_PTR

PUBLIC      I2C_ADDR_1
PUBLIC      BUF_LEN_1
PUBLIC      BUF_PTR_1
PUBLIC      I2C_ADDR_2
PUBLIC      BUF_LEN_2
PUBLIC      BUF_PTR_2

;-----
;* GLOBAL FUNCTION DECLARATION
;*
PUBLIC      _I2C_INIT_BYTE
PUBLIC      _I2C_TEST_DEVICE_BYTE
PUBLIC      _I2C_WRITE_BYTE
PUBLIC      _I2C_WRITE_SUB_BYTE
PUBLIC      _I2C_WRITE_SUB_SWINC_BYTE
PUBLIC      _I2C_WRITE_MEMORY_BYTE
PUBLIC      _I2C_WRITE_SUB_WRITE_BYTE
PUBLIC      _I2C_WRITE_COW_WRITE_BYTE
PUBLIC      _I2C_WRITE_REP_WRITE_BYTE
PUBLIC      _I2C_WRITE_REP_READ_BYTE
PUBLIC      _I2C_READ_BYTE
PUBLIC      _I2C_READ_STATUS_BYTE
PUBLIC      _I2C_READ_SUB_BYTE
PUBLIC      _I2C_READ_REP_READ_BYTE
PUBLIC      _I2C_READ_REP_WRITE_BYTE

;-----
;* GLOBAL FUNCTION DEFINITIONS
;*
_I2C_MCB_DATA SEGMENT DATA
RSEG         I2C_MCB_DATA

_I2C_INIT_BYTE:
    OWN_SLV_ADDR DS     1
    SLV_BUF_PTR DS     1

_I2C_TEST_DEVICE_BYTE:
_I2C_WRITE_BYTE:
_I2C_WRITE_SUB_BYTE:

```

```

12C WRITE SUB SWING BYTE:
12C WRITE MEMORY BYTE:
12C WRITE SUB WRITE BYTE:
12C WRITE SUB READ BYTE:
12C WRITE COM WRITE BYTE:
12C WRITE REF WRITE BYTE:
12C WRITE REF READ BYTE:
12C READ BYTE:
12C READ STATUS BYTE:
12C READ SUB BYTE:
12C READ REF READ BYTE:
12C READ REF WRITE BYTE:

      12C MCB:          DS          6
      I2C_ADDR 1        DATA      I2C_MCB+0
      BUF_LEN 1         DATA      I2C_MCB+1
      BUF_PTR 1         DATA      I2C_MCB+2
      I2C_ADDR 2        DATA      I2C_MCB+3
      BUF_LEN 2         DATA      I2C_MCB+4
      BUF_PTR 2         DATA      I2C_MCB+5

12C_STAT_DATA          SEGMENT DATA BITADDRESSABLE
RSEG                   I2C_STAT_DATA

I2C_CTRL:              DS          1
I2C_STAT:              DS          1

```


EIE/AN91007

EIE/AN91007

```

** TITLE (I2C_DATA_LOC) **
**
** INCLUDE FILE: I2C_DATA.LOC
** PACKAGE      : I2C
**
**
**-----**
**
** This file must be included into each I2C function,
** It contains the I2C Local symbol definitions
**
**-----**
**
** LOCAL SYMBOL DEFINITIONS
**
**
SDA      BIT      81H
SCL      BIT      80H

BUF_PTR   SET      R0
BUF_LEN   SET      R1
BIT_CNT   SET      R2
MESS_RETRY_CNT SET  R3
BUS_ERR_CLKS SET  R4
MEM_MESS_LEN SET  R5
MEM_DELAY_H SET  R6
MEM_DELAY_L SET  R7

I2C_LOCAL_SYMBOL_DEFINITIONS

```

```

$ TITLE (I2C_CODE.H)

*****
**
** INCLUDE FILE: I2C_CODE.H
** PACKAGE      : I2C
**
*****

**
** This file must be included into the ASSEMBLER MAIN
** It contains the EXTERNAL CODE references (Global
** function definitions) of the I2C functions
**
*****

***** GLOBAL FUNCTION DEFINITIONS *****
**

EXTRN CODE (I2C_INIT)
EXTRN CODE (I2C_TEST_DEVICE)
EXTRN CODE (I2C_WRITE)
EXTRN CODE (I2C_WRITE SUB)
EXTRN CODE (I2C_WRITE SUB SWINC)
EXTRN CODE (I2C_WRITE MEMORY)
EXTRN CODE (I2C_WRITE SUB WRITE)
EXTRN CODE (I2C_WRITE SUB READ)
EXTRN CODE (I2C_WRITE COM WRITE)
EXTRN CODE (I2C_WRITE REP WRITE)
EXTRN CODE (I2C_WRITE REP_READ)
EXTRN CODE (I2C_READ)
EXTRN CODE (I2C_READ STATUS)
EXTRN CODE (I2C_READ SUB)
EXTRN CODE (I2C_READ REP_READ)
EXTRN CODE (I2C_READ REP_WRITE)

*****

I2C_READ    EQU    00000000
I2C_WRITE   EQU    00000001
I2C_WRITE_CMD EQU    00000002

*****
**
** GLOBAL VARIABLE DEFINITIONS
**
*****

%NO_DEF EQU    00000000
%NO_READ EQU    00000001
%NO_WRITE EQU    00000002
%NO_CMD EQU    00000003
%NO_READ_1 EQU    00000004
%NO_READ_2 EQU    00000005
%NO_READ_3 EQU    00000006
%NO_READ_4 EQU    00000007
%NO_READ_5 EQU    00000008
%NO_READ_6 EQU    00000009
%NO_READ_7 EQU    00000010
%NO_READ_8 EQU    00000011
%NO_READ_9 EQU    00000012
%NO_READ_10 EQU    00000013
%NO_READ_11 EQU    00000014
%NO_READ_12 EQU    00000015
%NO_READ_13 EQU    00000016
%NO_READ_14 EQU    00000017
%NO_READ_15 EQU    00000018
%NO_READ_16 EQU    00000019
%NO_READ_17 EQU    00000020
%NO_READ_18 EQU    00000021
%NO_READ_19 EQU    00000022
%NO_READ_20 EQU    00000023
%NO_READ_21 EQU    00000024
%NO_READ_22 EQU    00000025
%NO_READ_23 EQU    00000026
%NO_READ_24 EQU    00000027
%NO_READ_25 EQU    00000028
%NO_READ_26 EQU    00000029
%NO_READ_27 EQU    00000030
%NO_READ_28 EQU    00000031
%NO_READ_29 EQU    00000032
%NO_READ_30 EQU    00000033
%NO_READ_31 EQU    00000034
%NO_READ_32 EQU    00000035
%NO_READ_33 EQU    00000036
%NO_READ_34 EQU    00000037
%NO_READ_35 EQU    00000038
%NO_READ_36 EQU    00000039
%NO_READ_37 EQU    00000040
%NO_READ_38 EQU    00000041
%NO_READ_39 EQU    00000042
%NO_READ_40 EQU    00000043
%NO_READ_41 EQU    00000044
%NO_READ_42 EQU    00000045
%NO_READ_43 EQU    00000046
%NO_READ_44 EQU    00000047
%NO_READ_45 EQU    00000048
%NO_READ_46 EQU    00000049
%NO_READ_47 EQU    00000050
%NO_READ_48 EQU    00000051
%NO_READ_49 EQU    00000052
%NO_READ_50 EQU    00000053
%NO_READ_51 EQU    00000054
%NO_READ_52 EQU    00000055
%NO_READ_53 EQU    00000056
%NO_READ_54 EQU    00000057
%NO_READ_55 EQU    00000058
%NO_READ_56 EQU    00000059
%NO_READ_57 EQU    00000060
%NO_READ_58 EQU    00000061
%NO_READ_59 EQU    00000062
%NO_READ_60 EQU    00000063
%NO_READ_61 EQU    00000064
%NO_READ_62 EQU    00000065
%NO_READ_63 EQU    00000066
%NO_READ_64 EQU    00000067
%NO_READ_65 EQU    00000068
%NO_READ_66 EQU    00000069
%NO_READ_67 EQU    00000070
%NO_READ_68 EQU    00000071
%NO_READ_69 EQU    00000072
%NO_READ_70 EQU    00000073
%NO_READ_71 EQU    00000074
%NO_READ_72 EQU    00000075
%NO_READ_73 EQU    00000076
%NO_READ_74 EQU    00000077
%NO_READ_75 EQU    00000078
%NO_READ_76 EQU    00000079
%NO_READ_77 EQU    00000080
%NO_READ_78 EQU    00000081
%NO_READ_79 EQU    00000082
%NO_READ_80 EQU    00000083
%NO_READ_81 EQU    00000084
%NO_READ_82 EQU    00000085
%NO_READ_83 EQU    00000086
%NO_READ_84 EQU    00000087
%NO_READ_85 EQU    00000088
%NO_READ_86 EQU    00000089
%NO_READ_87 EQU    00000090
%NO_READ_88 EQU    00000091
%NO_READ_89 EQU    00000092
%NO_READ_90 EQU    00000093
%NO_READ_91 EQU    00000094
%NO_READ_92 EQU    00000095
%NO_READ_93 EQU    00000096
%NO_READ_94 EQU    00000097
%NO_READ_95 EQU    00000098
%NO_READ_96 EQU    00000099
%NO_READ_97 EQU    00000100
%NO_READ_98 EQU    00000101
%NO_READ_99 EQU    00000102
%NO_READ_100 EQU    00000103
%NO_READ_101 EQU    00000104
%NO_READ_102 EQU    00000105
%NO_READ_103 EQU    00000106
%NO_READ_104 EQU    00000107
%NO_READ_105 EQU    00000108
%NO_READ_106 EQU    00000109
%NO_READ_107 EQU    00000110
%NO_READ_108 EQU    00000111
%NO_READ_109 EQU    00000112
%NO_READ_110 EQU    00000113
%NO_READ_111 EQU    00000114
%NO_READ_112 EQU    00000115
%NO_READ_113 EQU    00000116
%NO_READ_114 EQU    00000117
%NO_READ_115 EQU    00000118
%NO_READ_116 EQU    00000119
%NO_READ_117 EQU    00000120
%NO_READ_118 EQU    00000121
%NO_READ_119 EQU    00000122
%NO_READ_120 EQU    00000123
%NO_READ_121 EQU    00000124
%NO_READ_122 EQU    00000125
%NO_READ_123 EQU    00000126
%NO_READ_124 EQU    00000127
%NO_READ_125 EQU    00000128
%NO_READ_126 EQU    00000129
%NO_READ_127 EQU    00000130
%NO_READ_128 EQU    00000131
%NO_READ_129 EQU    00000132
%NO_READ_130 EQU    00000133
%NO_READ_131 EQU    00000134
%NO_READ_132 EQU    00000135
%NO_READ_133 EQU    00000136
%NO_READ_134 EQU    00000137
%NO_READ_135 EQU    00000138
%NO_READ_136 EQU    00000139
%NO_READ_137 EQU    00000140
%NO_READ_138 EQU    00000141
%NO_READ_139 EQU    00000142
%NO_READ_140 EQU    00000143
%NO_READ_141 EQU    00000144
%NO_READ_142 EQU    00000145
%NO_READ_143 EQU    00000146
%NO_READ_144 EQU    00000147
%NO_READ_145 EQU    00000148
%NO_READ_146 EQU    00000149
%NO_READ_147 EQU    00000150
%NO_READ_148 EQU    00000151
%NO_READ_149 EQU    00000152
%NO_READ_150 EQU    00000153
%NO_READ_151 EQU    00000154
%NO_READ_152 EQU    00000155
%NO_READ_153 EQU    00000156
%NO_READ_154 EQU    00000157
%NO_READ_155 EQU    00000158
%NO_READ_156 EQU    00000159
%NO_READ_157 EQU    00000160
%NO_READ_158 EQU    00000161
%NO_READ_159 EQU    00000162
%NO_READ_160 EQU    00000163
%NO_READ_161 EQU    00000164
%NO_READ_162 EQU    00000165
%NO_READ_163 EQU    00000166
%NO_READ_164 EQU    00000167
%NO_READ_165 EQU    00000168
%NO_READ_166 EQU    00000169
%NO_READ_167 EQU    00000170
%NO_READ_168 EQU    00000171
%NO_READ_169 EQU    00000172
%NO_READ_170 EQU    00000173
%NO_READ_171 EQU    00000174
%NO_READ_172 EQU    00000175
%NO_READ_173 EQU    00000176
%NO_READ_174 EQU    00000177
%NO_READ_175 EQU    00000178
%NO_READ_176 EQU    00000179
%NO_READ_177 EQU    00000180
%NO_READ_178 EQU    00000181
%NO_READ_179 EQU    00000182
%NO_READ_180 EQU    00000183
%NO_READ_181 EQU    00000184
%NO_READ_182 EQU    00000185
%NO_READ_183 EQU    00000186
%NO_READ_184 EQU    00000187
%NO_READ_185 EQU    00000188
%NO_READ_186 EQU    00000189
%NO_READ_187 EQU    00000190
%NO_READ_188 EQU    00000191
%NO_READ_189 EQU    00000192
%NO_READ_190 EQU    00000193
%NO_READ_191 EQU    00000194
%NO_READ_192 EQU    00000195
%NO_READ_193 EQU    00000196
%NO_READ_194 EQU    00000197
%NO_READ_195 EQU    00000198
%NO_READ_196 EQU    00000199
%NO_READ_197 EQU    00000200
%NO_READ_198 EQU    00000201
%NO_READ_199 EQU    00000202
%NO_READ_200 EQU   
```

EIE/A191007

```
*-----*
* $TITLE(I2C_Init command)
*-----*
*
* SOURCE FILE : I2C_INIT.ASM
* PACKAGE      : I2C-
*-----*
$DEBUG

*-----*
* INCLUDES
*-----*
*
* LOCAL SYMBOL DECLARATIONS
*-----*
RETRIES          SET      R3

$NOLIST
$INCLUDE(REG751.H)
$INCLUDE(I2C_DATA.GLO)
$INCLUDE(I2C_DATA.LOC)
$LIST

*-----*
* GLOBAL FUNCTION DEFINITIONS
*-----*
PUBLIC I2C_INIT

*-----*
* CODE SEGMENT
*-----*
I2C DRIVER SEGMENT CODE
RSEG I2C_DRIVER

;MPF:::I2C::I2C_INIT.ASM:I2C_INIT
*
* FUNCTION NAME:         I2C_INIT
* PACKAGE:               I2C-
* DESCRIPTION:
*   Initialize I2C interface: set SDA & SCL, enable time out
*   timer, allow 16 MHz (CPL.CT0 = 0). Set the number of
*   retries (max 7) into the I2C_STAT. Bit 7,6 and 5 of the
*   I2C_STAT contain the number of retries. Those bits may
*   not be changed during the IyC routines.
* INPUT:
*   Before calling I2C_INIT the main program must take care
*   that the correct parameters are available in
*   OWN_SIV_ADDR, SIV_BUF_PTR and I2C_INIT_BYTE+2, this
*   is done automatically when using C, PL/M or the pre-
*   defined assembler macro (available in I2C_MAC.DEF)
* OUTPUT:
*   Initialized IyC and retry number in I2C_STAT 7..5
*EMPF
_I2C_INIT:
MOV     I2CFG,#I2C_ENABLE ;CLR TIRUN, CLR MASTRO
SETB    EIT
SETB    EIT
SETB    EA                ;enable interrupts
```

[illegible]

```

$TITLE(I2C age Handler)
;-----
;
; SOURCE FILE : I2C HAND.ASM
; PACKAGE      : I2C
;-----
$DEBUG

;-----
; INCLUDES
;-----
$NOLIST
$INCLUDE(REG751.H)
$INCLUDE(I2C_DATA.GLO)
$INCLUDE(I2C_DATA.LOC)
$LIST

;-----
; GLOBAL REFERENCES
;-----
EXTRN CODE(I2C_STOP)
EXTRN CODE(I2C_TRX_BYTE)
EXTRN CODE(I2C_TRX_ADDR)
EXTRN CODE(I2C_RCV_BYTE)
EXTRN CODE(I2C_TRX_BLOCK)
EXTRN CODE(I2C_RCV_BLOCK)
EXTRN CODE(I2C_STRT_SLVAD)
EXTRN CODE(I2C_RSTRT_SLVAD)

;-----
; GLOBAL FUNCTION DEFINITIONS
;-----
PUBLIC I2C_MESS_HAND

;-----
; LOCAL SYMBOL DECLARATIONS
;-----
RWN      BIT      OE0H      ;bit ACC.0
I2C_PSW  EQU      8

;-----
; CODE SEGMENT
;-----
I2C_DRIVER_SEGMENT_CODE
RSEG I2C_DRIVER

;-----
;MPF:::I2C::I2C_HAND.ASM:I2C_MESS_HAND
;-----
; FUNCTION NAME:      I2C_HAND
; PACKAGE:           I2C
; DESCRIPTION:
;   Transmit an I2C age, includes error handling
; INPUT:  age control byte I2C_CTRL (bit addressable)
;         age control block I2C_MCB, containing:
;         I2C_ADDR1 (i.e. slave address)
;         BUF_LEN1 (i.e. number of bytes to trx.)
;         BUF_PTR1 (i.e. transmit buffer)
;         I2C_ADDR2 (i.e. sub address)

```

```

;
; BUF_LEN2 (i.e. length of second data blk)
; BUF_PTR2 (i.e. second transmit buffer)
;
; OUTPUT: I2C_ERROR byte (bit addressable)
;
;EMP-----
I2C_MESS_HAND:
    PUSH    PSW,#I2C_PSW      ;set RBL
    ANL     I2C_STAT,#07H     ;clr all but retry bits
    MOV     MESS_RETRY_CNT,I2C_STAT
    INC     MESS_RETRY_CNT     ;load retry counter
RETRY:
    ANL     I2C_STAT,#07H     ;clr all but retry bits
    MOV     A,I2C_ADDR_1      ;load SLV_ADDR
    CLR     RWN               ;if (subaddress)
    JB      ADDR2_SUB,STRT     ; RWN = 0
    MOV     C,RWN_BLK1        ;else
    MOV     RWN,C             ; RWN = RWN_BLK1
STRT:
    ACALL   I2C_STRT_SLVAD     ;send START+SLV_ADDR+RWN
    JNB     I2C_ERR,CONTINUE   ;branch offset to large
    AJMP    EXIT
CONTINUE:
    JB      TEST_DEVICE,M_STOP
    MOV     BUF_PTR,BUF_PTR_1 ;load pointer block1
    MOV     BUF_LEN,BUF_LEN_1 ;load length block1
    JNB     ADDR2_SUB,BLOCK    ;if (addr2 sub)
    MOV     A,I2C_ADDR_2      ;load sub address
    ACALL   I2C_TRX_BYTE      ;trx_byte(sub address)
    JB      I2C_ERR,EXIT      ;if (error) exit();
    JNB     REP_STRT_BLK1,BLOCK ; if (rep. start blk1)
    MOV     A,I2C_ADDR_1      ;load slave address
    SETB    RWN               ;read
    ACALL   I2C_RSTRT_SLVAD    ; send RSTART+SLV_ADDR
    JB      I2C_ERR,EXIT      ; if (error) exit();
BLOCK:
    JNB     RWN_BLK1,TRX_1     ;if ((rwn blk1) == read)
    ACALL   I2C_RCV_BLOCK      ; rcv_block(&_datal,cnt1)
    SJMP    END_BLOCK1
TRX_1:
    ACALL   I2C_TRX_BLOCK      ; trx_block(&datal,cnt1)
END_BLOCK1:
    JB      I2C_ERR,EXIT      ;if (error) exit();
    JNB     BLOCK2,M_STOP     ;if (2nd block of data)
    MOV     BUF_PTR,BUF_PTR_2 ;{
    MOV     BUF_LEN,BUF_LEN_2 ;{
    JNB     ADDR2_DATA2        ; if (addr2)
    JNB     REP_STRT_BLK2,DATA2 ; if (rep. start blk2)
    MOV     A,I2C_ADDR_2      ; { set address2
    JNB     ADDR2_SUB,SET_RWN  ; if(addr2 sub)
    MOV     A,I2C_ADDR_1      ; set address1
    SET_RWN:
    MOV     C,RWN_BLK2        ; /* same slave */
    MOV     RWN,C             ; modify RWN_BLK1
    ACALL   I2C_RSTRT_SLVAD    ; send RSTART+SLV_ADDR
    JB      I2C_ERR,EXIT      ; if (error) exit();
DATA2:
    JNB     RWN_BLK2,TRX_2     ; if ((rwn blk2) == read)
    ACALL   I2C_RCV_BLOCK      ; rcv_block(&_dat1,c1)
    SJMP    BLOCK_ERR
TRX_2:
    ACALL   I2C_TRX_BLOCK      ; else
    BLOCK_ERR:
    JB      I2C_ERR,EXIT      ; if (error) exit();
M_STOP:

```

I²C driver routines for 8XC751/2 microcontrollers

```

ACALL I2C_STOP ;}
JB I2C_ERR,EXIT ; if (error) exit();
AJMP RESTORE_CONTEXT

;*MPF:::I2C::I2C_HAND.ASM:EXIT-----*
;* FUNCTION NAME: EXIT I2C HAND *
;* PACKAGE: I2C *
;* DESCRIPTION: *
;* Exit an I2C message. This routine is only entered if an *
;* I2C error has occurred. If more retries must be made, *
;* the message is started again. If no retry must be made, *
;* the message handler is left after setting the carry. *
;* Carry is 1 indicates that an error has occurred (return *
;* value for C and PL/M calls). If the routine is entered *
;* at the RESTORE_CONTEXT label, no error has occurred *
;* OUTPUT: I2C_ERROR byte (bit addressable) *
;* *
;*EMP-----*
EXIT:
JNB TIME_ERR,TO_RETRY
JNB BUS_RECOVERED,RESTORE_CONTEXT
TO_RETRY:
MOV I2CON,#I2C_RELEASE
DJNZ MESS_RETRY_CNT,RETRY ; if (no more retries)

RESTORE_CONTEXT:
MOV I2CFG,#I2C_ENABLE ; SILVEN=1,MSTRQ=0,TIRN=0
MOV A,I2C_STAT
POP PSW ; restore PSW
MOV C,I2C_ERR
END MESSAGE:
SETB EI2 ;label for debugging
RET ; with XRAY

;* HISTORY *
;* *
;* 12-06-91 J.C. Pijnenburg original version *
;* *
;* END *

```

```

$TITLE(I2C_Basic Functions)
;*
;* SOURCE FILE : I2C_BASI.ASM
;* PACKAGE : I2C
;*
$DEBUG

;* INCLUDES *
;*
$NOLIST
$INCLUDE(REG751.H)
$INCLUDE(I2C_DATA.GLO)
$INCLUDE(I2C_DATA.LOC)
$LIST

;* GLOBAL REFERENCES *
;*
EXTRN CODE(I2C_SLV_TRK)
EXTRN CODE(I2C_SLV_RCV)
EXTRN CODE(ADDR_RECOG)

;* GLOBAL FUNCTION DEFINITIONS *
;*
PUBLIC I2C_STRT_SLVAD
PUBLIC I2C_RSTRT_SLVAD
PUBLIC I2C_STOP
PUBLIC I2C_TRX_BYTE
PUBLIC I2C_TRX_ADDR
PUBLIC I2C_RCV_BYTE
PUBLIC I2C_RCV_ADDR
PUBLIC I2C_TRK_BLOCK
PUBLIC ADDR_COMPARE

;* INTERRUPT CODE SEGM: TIMER1 *
;*
CSEG AT 01BH

;*MPF:::I2C::I2C_INIT.ASM:I2C_TIME_OUT-----*
;* FUNCTION NAME: I2C_TIME_OUT *
;* PACKAGE: I2C *
;* DESCRIPTION: *
;* I2C time out routine, clear timer interrupt, set the *
;* TIME_ERR bit. If the 8xC751 was I2C bus master while the *
;* interrupt occurred (RECOVER = 1), an attempt to recover *
;* the bus is made. *
;* To recover, SDA and SCL are set, if SCL remains low, the *
;* I2C bus cannot be recovered and the routine is left. *
;* If SCL is HIGH but SDA is low, 9 additional clocks are *
;* generated. If SDA becomes HIGH, a STOP is made *
;* If the 8xC751 was not I2C bus master (RECOVER = 0), the *
;* bus is released. *

```

I2C driver routines for 8XC751/2 microcontrollers 8XC751/2 EIE/AN91007

```

; *EMP-----
SETB  CLR TI
SETB  TIME_ERR
CLR    TIRUN
AJMP   TI_INT

; *-----
; * CODE SEGMENT
; *-----
; * I2C DRIVER SEGMENT CODE
; * RSEG I2C_DRIVER

; *MPF:::I2C::I2C_BAS-----
; *
; * SOURCE FILE:      I2C_BAS.ASM
; * PACKAGE:          I2C
; * DESCRIPTION: This file contains the basic I2C functions
; * being: START, REP START, STOP, TRX SLV ADDR, TRX BYTE
; * TRX BLOCK, RCV BYTE, RCV BLOCK. The wait loops on ATN
; * are left if the ATN bit is set or if a TI interrupt
; * occurs (time out). In case of a time out the program
; * checks whether it is a "real" time out (max. time is
; * exceeded). If yes, the program continues and will enter
; * a error routine, if no the ATN wait loop is reentered
; *
; *EMP-----

; *MPF:::I2C::I2C_BAS.ASM:I2C_START-----
; *
; * FUNCTION NAME:      I2C_START
; * PACKAGE:            I2C
; * DESCRIPTION:
; * Generate a start condition on the I2C bus, Set the
; * MASTRQ bit. If 8XC751 has not become master on ATN,
; * switch to receive mode and check if the own slave
; * address is received.
; *
; * INPUT: none
; * OUTPUT: I2C_ERROR (0, no error; 1, error)
; * OUTPUT CONDITION: SCL is stretched
; *
; *EMP-----
I2C_STRT SLVAD:
SETB  TIRUN
JB    STR,IS_MASTER ; already started
CLR    EI2 ; disable I2C interrupt
MOV    I2CFG,#I2C_START_CTRL
ACALL  WAIT_ATN
IS_MASTER:
JB    MASTER,I2C_TRX_ADDR
MOV    I2CON,#C_STR
ACALL  I2C_RCV_ADDR
JB    I2C_ERR,END_I2C_START
ACALL  ADDR_COMPARE
START_ERR:
SETB  I2C_ERR
END_I2C_START:
RET

; *MPF:::I2C::I2C_BAS.ASM:I2C_STOP-----
; *
; * FUNCTION NAME:      I2C_STOP
; * PACKAGE:            I2C
; * DESCRIPTION:
; * Generate a stop condition on the I2C bus
; * The STOP condition is generated by setting the XSTP bit.
; * If no error occurs, this function is left with I2C bus
; * released and TI stopped. In case of an error the bus is
; * released in the message handler.
; *
; * INPUT: none
; * OUTPUT: I2C_ERROR (0, no error; 1, error)
; *
; *EMP-----
I2C_STOP:
CLR    MASTRQ
MOV    I2CON,#S_STP
ACALL  WAIT_ATN ; wait for rising SCL
JNB    DRDY,I2C_BASIC_ERR
MOV    I2CON,#C_DRDY
ACALL  WAIT_ATN ; wait for stop
MOV    I2CON,#I2C_RELEASE
CLR    TIRUN
RET

; *MPF:::I2C::I2C_BAS.ASM:I2C_TRX_ADDR-----
; *
; * FUNCTION NAME:      I2C_TRX_ADDR
; * PACKAGE:            I2C
; * DESCRIPTION:
; * This function calls I2C_TRX_BYTE to transmit the
; * slave address, if an arbitration is lost before the last
; * bit is transmitted, the function receives the remaining
; * bits (receive mode), and checks whether the own slave
; * address has been received (call ADDR_CMP).
; *
; * INPUT: byte to transmit in ACC
; * OUTPUT: I2C_ERROR (0, no error; 1, error)
; * OUTPUT CONDITION: SCL is stretched
; *
; *EMP-----

```

```

; * FUNCTION NAME:      I2C_TRX_ADDR
; * PACKAGE:            I2C
; * DESCRIPTION:
; * Generate a repeated start condition on the I2C bus
; * The repeated start is generated by setting the XSTR
; * bit. If STR is not set (by hardware), the I2C bus is
; * released, no check for own slave address is done after a
; * repeated start.
; *
; * INPUT: none
; * OUTPUT: I2C_ERROR (0, no error; 1, error)
; * OUTPUT CONDITION: SCL is stretched
; *
; *EMP-----
I2C_RSTR SLVAD:
MOV    I2CON,#S_RSTR
ACALL  WAIT_ATN
JNB    DRDY,I2C_BASIC_ERR
MOV    I2CON,#C_DRDY
ACALL  WAIT_ATN
JNB    STR,I2C_BASIC_ERR
JMP    I2C_TRX_ADDR

; *MPF:::I2C::I2C_BAS.ASM:I2C_STOP-----
; *
; * FUNCTION NAME:      I2C_STOP
; * PACKAGE:            I2C
; * DESCRIPTION:
; * Generate a stop condition on the I2C bus
; * The STOP condition is generated by setting the XSTP bit.
; * If no error occurs, this function is left with I2C bus
; * released and TI stopped. In case of an error the bus is
; * released in the message handler.
; *
; * INPUT: none
; * OUTPUT: I2C_ERROR (0, no error; 1, error)
; *
; *EMP-----
I2C_STOP:
CLR    MASTRQ
MOV    I2CON,#S_STP
ACALL  WAIT_ATN ; wait for rising SCL
JNB    DRDY,I2C_BASIC_ERR
MOV    I2CON,#C_DRDY
ACALL  WAIT_ATN ; wait for stop
MOV    I2CON,#I2C_RELEASE
CLR    TIRUN
RET

; *MPF:::I2C::I2C_BAS.ASM:I2C_TRX_ADDR-----
; *
; * FUNCTION NAME:      I2C_TRX_ADDR
; * PACKAGE:            I2C
; * DESCRIPTION:
; * This function calls I2C_TRX_BYTE to transmit the
; * slave address, if an arbitration is lost before the last
; * bit is transmitted, the function receives the remaining
; * bits (receive mode), and checks whether the own slave
; * address has been received (call ADDR_CMP).
; *
; * INPUT: byte to transmit in ACC
; * OUTPUT: I2C_ERROR (0, no error; 1, error)
; * OUTPUT CONDITION: SCL is stretched
; *
; *EMP-----

```


I²C driver routines for 8XC751/2 microcontrollers (8XC751/2) EIE/AN91007

```

; *EMP-----
I2C_TRX_ADDR:
    ACALL I2C_TRX_BYTE
    JNB I2C_ERR,END_TRX_ADDR
    DJNZ BIT_CNT,CONTINUE
    AJMP END_TRX_ADDR
CONTINUE:
    JNB ARL,END_TRX_ADDR
    JB STR,END_TRX_ADDR ;parasitaire START
    JB STP,END_TRX_ADDR ;parasitaire STOP
    CLR I2C_ERR
    INC BIT_CNT
    CLR OEOH ;RDAT = 0 to ACC.0
RCV_NEXT_BIT:
    MOV I2CON,%C_XMTA+C_DRDY+C_ARL ;rcv mode
    ACALL WAIT_ATN
    JNB DRDY,RESTORE_ERR
    MOV C,RDAT
    RLC A
    DJNZ BIT_CNT,RCV_NEXT_BIT
    ACALL ADDR_COMPARE
RESTORE_ERR:
    SETB I2C_ERR ;set err for ret main
END_TRX_ADDR:
    RET

; *MPF:::I2C::I2C_BASI.ASM:I2C_ADDR_COMPARE-----
; *
; * FUNCTION NAME: I2C_ADDR_COMPARE
; * PACKAGE: I2C
; * DESCRIPTION:
; * Compares the contents of the accumulator (received
; * address) with the OWN SLV ADDR. If equal and the RWN
; * bit is 0 (master transmit, slave receive) I2C_SLV_RCV is
; * called. If equal and the RWN bit is 1 (master receive,
; * slave transmit) I2C_SLV_TRX is called. If not equal exit*
; * I2C_ADDR_COMPARE
; *
; * INPUT: received address in ACC
; * OUTPUT: I2C ERROR (0, no error; 1, error)
; * OUTPUT CONDITION: SCL is stretched
; *
; *EMP-----
ADDR_COMPARE:
    XRL A,OWN_SLV_ADDR
    JZ SEND_ACK
    CJNE A,%1,END_ADDR_COMPARE
SEND_ACK:
    MOV I2DAT,%0
    ACALL WAIT_ATN
    JNB DRDY,END_ADDR_COMPARE
    JZ SLAVE_RCV
    AJMP I2C_SLV_TRX
SLAVE_RCV:
    AJMP I2C_SLV_RCV
END_ADDR_COMPARE:
    RET

; *MPF:::I2C::I2C_BASI.ASM:I2C_BASIC_ERR-----
; *
; * FUNCTION NAME: I2C_BASIC_ERR
; * PACKAGE: I2C

```

```

; * DESCRIPTION:
; * Set the I2C_ERR bit. The message handler tests this bit
; *
; * INPUT: none
; * OUTPUT: I2C_ERROR = 1
; *
; *EMP-----
I2C_BASIC_ERR:
    SETB I2C_ERR
    RET

; *MPF:::I2C::I2C_BASI.ASM:I2C_TRX_BYTE-----
; *
; * FUNCTION NAME: I2C_TRX_BYTE
; * PACKAGE: I2C
; * DESCRIPTION:
; * Transmit a byte over the I2C bus.
; * NOTE: The STR bit is cleared here in stead of in the
; * I2C START routine, because there must be valid
; * data in I2DAT before STR may be cleared (also
; * releases the SCL line).
; *
; * The I2C_TRX_BYTE function transmits a byte over the I2C
; * bus, after the last bit has been transmitted,
; * the function switches to receive mode to receive the
; * acknowledge bit. If NACK is received, the NO_ACK bit is
; * set. If arbitration is lost or an error occurs during
; * I2C_TRX_BYTE the function is exit with the I2C_ERR bit
; * set.
; *
; * INPUT: byte to transmit in ACC
; * OUTPUT: I2C ERROR (0, no error; 1, error)
; * OUTPUT CONDITION: SCL is stretched
; *
; *EMP-----
I2C_TRX_BYTE:
    MOV BIT_CNT,%8
TRX_BIT:
    MOV I2DAT,A
    MOV I2CON,%C_STRT ;release SCL
    ;if STR clear STR
    ;else dummy MOV
    ACALL WAIT_ATN
    JNB DRDY,I2C_BASIC_ERR
    RL A
    DJNZ BIT_CNT,TRX_BIT
    MOV I2CON,%C_XMTA+C_DRDY ;receive mode
    ACALL WAIT_ATN
    JNB DRDY,I2C_BASIC_ERR
    JNB RDAT,TRX_BYTE_RDY ;stretch SCL
    SETB NO_ACK
TRX_BYTE_RDY:
    RET

; *MPF:::I2C::I2C_BASI.ASM:I2C_RCV_BYTE-----
; *
; * FUNCTION NAME: I2C_RCV_BYTE
; * PACKAGE: I2C
; * DESCRIPTION:
; * Receive a byte from the I2C bus
; * This is one function which receives a byte into acc.
; * RCV_BYTE first releases the SCL and then receives the 8
; * bits. If RCV_ADDR is called, the first bit is already in
; * the RDAT register, this must first be saved before the
; * SCL is released.

```

I2C driver routines for 8XC751/2 microcontrollers

```

; * FUNCTION NAME: I2C
; * INPUT: none
; * OUTPUT: I2C_ERROR (0, no error; 1, error)
; * if (! I2C_ERROR) received byte in ACC.
; *
; *-----*
I2C_RCV_BYTE:
MOV I2CON, #C_XMTA+C_DRDY ;rel. SCL, rcv mode
I2C_RCV_ADDR:
MOV BIT_CNT, #8
CLR A ; rcv first bit
RCV_BIT:
ACALL WAIT_ATN
JNB DRDY, I2C_BASIC_ERR
DJNZ BIT_CNT, NOT_LAST_BIT
MOV C, RDAT
RLC A
NOT_LAST_BIT:
ORL A, I2DAT ; save bit, rel. SCL
RL A
SJP RCV_BIT
; *-----*
; *MPF:::I2C:::I2C_BASI.ASM:I2C_TRX_BLOCK-----*
; *
; * FUNCTION NAME: I2C_TRX_BLOCK
; * PACKAGE: I2C
; * DESCRIPTION:
; * Transmit a block of bytes over the I2C bus
; * The I2C_TRX_BLOCK function transmits as much bytes as
; * defined in BUF_LEN (set R2), before the message handler
; * is called, BUF_LEN_1 or BUF_LEN_2 is copied into BUF_LEN
; *
; * INPUT: pointer to begin of block data in BUF_PTR (R0)
; * byte counter in BUF_LEN (R2)
; * OUTPUT: I2C_ERROR (0, no error; 1, error)
; *
; *-----*
I2C_TRX_BLOCK:
MOV A, #BUF_PTR ;load byte
ACALL I2C_TRX_BYTE
JB I2C_ERR, END_TRX_BLOCK
JB NO_ACK, END_TRX_BLOCK
INC BUF_PTR
DJNZ BUF_LEN, I2C_TRX_BLOCK
END_TRX_BLOCK:
RET
; *-----*
; *MPF:::I2C:::I2C_BASI.ASM:I2C_RCV_BLOCK-----*
; *
; * FUNCTION NAME: I2C_RCV_BLOCK
; * PACKAGE: I2C
; * DESCRIPTION:
; * Receive a block of bytes from the I2C bus
; * The I2C_RCV_BLOCK function receives as much bytes as
; * defined in BUF_LEN (set R2), before the message handler
; * is called, BUF_LEN_1 or BUF_LEN_2 is copied into BUF_LEN
; *
; * INPUT: pointer to begin of receive buffer BUF_PTR (R0)
; * byte counter in BUF_LEN (R2)
; * OUTPUT: I2C_ERROR (0, no error; 1, error)
; * if (!I2C_ERROR) received bytes in buffer.
; *
; *-----*

```

```

ACK_RCV_BYTE:
MOV I2DAT, #0 ;send ACK
ACALL WAIT_ATN
JNB DRDY, I2C_BASIC_ERR
I2C_RCV_BLOCK:
ACALL I2C_RCV_BYTE
JB I2C_ERR, END_RCV_BLOCK
INC BUF_PTR, A ;save byte
DJNZ BUF_LEN, ACK_RCV_BYTE
MOV I2DAT, #80H ;send NACK
ACALL WAIT_ATN
JNB DRDY, I2C_BASIC_ERR
END_RCV_BLOCK:
RET
; *-----*
; *MPF:::I2C:::I2C_BASI.ASM:WAIT_ATN-----*
; *
; * FUNCTION NAME: WAIT_ATN
; * PACKAGE: I2C
; * DESCRIPTION:
; * The WAIT_ATN function waits for the ATN bit to be set.
; * The function is left if the ATN bit is set or if the
; * TIME_ERR bit is set. The TIME_ERR bit indicates that a
; * bus timeout has occurred. If the 8XC751 enters this
; * function as a master, the RECOVER bit is set,
; * indicating that in case of a timeout, a bus recover
; * action must be started.
; *
; *-----*
; *MPF:::I2C:::I2C_BASI.ASM:TI_INT-----*
; *
; * FUNCTION NAME: TI_INT
; * PACKAGE: I2C
; * DESCRIPTION:
; * The TI_INT handles the timeout interrupt. It is
; * entered when a time out occurs (during WAIT_ATN).
; * The function is placed here to be sure that it is
; * linked when placed in a library.
; *
; *-----*
; *MPF:::I2C:::I2C_BASI.ASM:RETRY_LOOP-----*
; *
; * FUNCTION NAME: RETRY_LOOP
; * PACKAGE: I2C
; * DESCRIPTION:
; * The RETRY_LOOP function handles the timeout interrupt. It is
; * entered when a time out occurs (during WAIT_ATN).
; * The function is placed here to be sure that it is
; * linked when placed in a library.
; *
; *-----*

```

¹²C driver routines for 8XC751/2 microcontrollers EIE/AN91007

```

CLR          SCL_DELAY
ACALL        SCL
SETB         SCL
ACALL        SCL_DELAY
JB           SDA_MAKE_STOP      ;if SDA = 1 make stop
D.JNZ       BUS_ERR_CLKS,RETRY_LOOP
RETI

MAKE_STOP:   CLR          SCL
NOP
NOP
ACALL        SDA
ACALL        SCL_DELAY
SETB         SCL
ACALL        SCL_DELAY
ACALL        SDA                ;make stop condition
SETB         BUS_RECOVERED

RETRY_INT:   RETI

SCL_DELAY:   ;delay of 9 periods (>= 6 micro sec.)
NOP          ; ACALL(2) + 5 NOP (4) + RET (2)
NOP
NOP
NOP
NOP
RET

*****
** H I S T O R Y **
*****
03-07-91 J.C. Pijenburg original version
*****
END

```

```

$TITLE(I2C_Test_Device command)
;-----*
;
; SOURCE FILE : I2C_TDEV.ASM
; PACKAGE : I2C
;-----*
$DEBUG

;-----*
; INCLUDES
;-----*
$NOLIST
$INCLUDE(I2C_DATA.GLO)
$LIST

;-----*
; GLOBAL REFERENCES
;-----*
EXTRN CODE(I2C_MESS_HAND)

;-----*
; GLOBAL FUNCTION DEFINITIONS
;-----*
PUBLIC _I2C_TEST_DEVICE

;-----*
; LOCAL SYMBOL DECLARATIONS
;-----*
TEST_DEVICE_MASK EQU 80H
; REP STRT BLK1 = 0 (NO)
; RWN BLK1 = 0 (WRITE)
; ADDR2 = 0 (NO)
; ADDR2 SUB = 0 (--)
; BLOCK2 = 0 (NO)
; RWN BLK2 = 0 (--)
; REP STRT BLK2 = 0 (--)
; T_DEVICE = 1 (--)

;-----*
; CODE SEGMENT
;-----*
I2C DRIVER SEGMENT CODE
RSEG I2C_DRIVER

;MPF:::I2C::I2C_TDEV.ASM:I2C_TEST_DEVICE
;
; FUNCTION NAME: I2C_TEST_DEVICE
; PACKAGE: I2C
; DESCRIPTION:
; Address a slave , if ack received slave was present
;
; PROTOCOL:
; <S><SLV_ADDR><W><A><P>
;
; INPUT: Message control byte I2C_CTRL (bit addressable)
; Message control block I2C_MCB, containing:
; I2C_ADDR1 (slave address)
;

```

```

; * OUTPUT: I2C_ERROR byte (bit addressable)
; *
; *EMP
;-----*
_I2C_TEST_DEVICE:
MOV I2C_CTRL,#TEST_DEVICE_MASK
AJMP I2C_MESS_HAND
END

```

EIE/AN91007

```

**TITLE(I2C_Write command)
**
**SOURCE FILE : I2C_WRITE.ASM
**PACKAGE      : I2C
**
$DEBUG
**
**-----
** INCLUDES
**-----
$NOLIST
$INCLUDE(I2C_DATA.GLO)
$LIST
**
**-----
** GLOBAL REFERENCES
**-----
EXTRN CODE(I2C_MESS_HAND)
**
**-----
** GLOBAL FUNCTION DEFINITIONS
**-----
PUBLIC _I2C_WRITE
**
**-----
** LOCAL SYMBOL DECLARATIONS
**-----
WRITE_MASK EQU 00H
;RPN_STRT_BLK1 = 0 (NO)
;RW_N_BLK1 = 0 (WRITE)
;ADDR2 = 0 (NO)
;ADDR2_SUB = 0 (--)
;BLOCKZ = 0 (NO)
;RW_N_BLK2 = 0 (--)
;REP_STRT_BLK2 = 0 (--)
;TEST_DEVICE = 0 (--)
**
**-----
** CODE SEGMENT
**-----
I2C_DRIVER_SEGMENT_CODE
RSEG I2C_DRIVER
**
**MPF::I2C::I2C_WRITE.ASM:I2C_WRITE
**
**FUNCTION NAME: I2C_WRITE
**PACKAGE: I2C
**DESCRIPTION:
Write n bytes to a slave device.
**
**PROTOCOL:
<S><SLV_ADDR><W><A><D0><A><D1><A>..<Dn-1><A><P>
**
**INPUT: Message control byte I2C_CTRL (bit addressable)
Message control block I2C_MCB containing:
I2C_ADDR1 (slave address)
BUF_LEN1 (number of bytes (n) to tx.)
BUF_PTR1 (ptr to transmit buffer)

```

```
*  
;* OUTPUT: I2C_ERROR byte   (bit addressable)  
;  
;-----  
;*EMP  
  
_I2C_WRITE:  
MOV     I2C_CTRL,#WRITE_MASK  
AJMP    I2C_MESS_HAND  
  
END
```

I2C driver routines for 8XC751/2 microcontrollers EIE/AN91007

```
*
*
* $TITLE(I2C_Write_Sub command)
*
*
* SOURCE FILE : I2C_WSUB.ASM
* PACKAGE : I2C
*
* $DEBUG
*
*
* INCLUDES
*
* $NOLIST
* $INCLUDE(I2C_DATA.GLO)
* $LIST
*
*
* GLOBAL REFERENCES
*
* EXTRN CODE(I2C_MESS_HAND)
*
*
* GLOBAL FUNCTION DEFINITIONS
*
* PUBLIC _I2C_WRITE_SUB
*
*
* LOCAL SYMBOL DECLARATIONS
*
*
* WRITE_SUB_MASK EQU 0CH
* ;REP_STRT_BLK1 - 0 (NO)
* ;RWN_BLK1 - 0 (WRITE)
* ;ADDR2 - 1 (YES)
* ;ADDR2_SUB - 1 (YES)
* ;BLOCK2 - 0 (NO)
* ;RWN_BLK2 - 0 (--)
* ;REP_STRT_BLK2 - 0 (--)
* ;TEST_DEVICE - 0 (--)
*
*
* CODE SEGMENT
*
* I2C DRIVER SEGMENT CODE
* RSEG I2C_DRIVER
*
*
* *MPF:::I2C::I2C_WSUB.ASM:I2C_WRITE_SUB
*
* FUNCTION NAME: I2C_WRITE_SUB
* PACKAGE: I2C
* DESCRIPTION:
* Write a block of data (a length n) preceded by a
* sub address to a slave device.
*
* PROTOCOL:
* <S><SLV_ADDR><W><A><SUB_ADDR><A><Da0><A><Da1><A>...<A>
* <Dan-1><A><CP>
*
* INPUT: Message control byte I2C_CTRL (bit addressable)
* Message control block I2C_MCB, containing:
```

```
*
* I2C_ADDR1 (slave address)
* BUF_LEN1 (number of bytes in block)
* BUF_PTR1 (ptr to block)
* I2C_ADDR2 (sub address)
*
* OUTPUT: I2C_ERROR byte (bit addressable)
*
* *EMP
*
* _I2C_WRITE SUB:
* MOV I2C_CTRL,#WRITE_SUB_MASK
* AJMP I2C_MESS_HAND
*
* END
```


I2C driver routines for 8XC751/2 microcontrollers

```

* I2C driver routines for 8XC751/2 microcontrollers
*
* SOURCE FILE : I2C_WSWI.ASM
* PACKAGE : I2C
*
* INCLUDES
*
* GLOBAL REFERENCES
*
* GLOBAL FUNCTION DEFINITIONS
*
* LOCAL SYMBOL DECLARATIONS
*
* WRITE SUB SWINC MASK EQU 00CH
* WRITE MEMORY MASK EQU 02CH
*
* ;RWN_BLK1 = 0 (NO)
* ;ADDR2 = 1 (YES)
* ;ADDR2 SUB = 1 (YES)
* ;BLOCK2 = 0 (NO)
* ;RWN_BLK2 = 0 or 1 (no blk2,
* ; used for delay/no delay)
* ;REP_STRT_BLK2 = 0 (--)
* ;TEST_DEVICE = 0 (--)
*
* CODE SEGMENT
*
* I2C DRIVER SEGMENT CODE
* RSEG I2C_DRIVER
*
* *MPF::I2C:I2C_WSWI.ASM:I2C_WRITE_SUB_SWINC
*
* * FUNCTION NAME: I2C_WRITE_SUB_SWINC
* * PACKAGE: I2C
* * DESCRIPTION:
* * Transmit an I2C message, the message is split into
* * sub messages. Each sub message transmits one byte.
* * If the slave is an EEPROM, a delay is generated after
* * each sub message. The RWN_BLK2 is not used in the

```

```

* message handler (no block 2) and is therefore free to
* distinguish between write to EEPROM (1=delay) and other
* (0= no delay)
*
* PROTOCOL:
* <S><SLV_ADDR><W><A><SUB_ADDR><A><D0><A><P>
* if (RWN_BLK2) delay 40 ms
* <S><SLV_ADDR><W><A><SUB_ADDR+1><A><D1><P>
* if (RWN_BLK2) delay 40 ms
* <- if (RWN_BLK2) delay 40 ms
* <S><SLV_ADDR><W><A><SUB_ADDR+n-1><A><Dn-1><P>
* if (RWN_BLK2) delay 40 ms
*
* INPUT: Message control byte I2C_CTRL (bit addressable)
* Message control block I2C_MCB, containing:
* I2C_ADDR1 (slave address)
* BUF_LEN1 (number of bytes (mess) to trx.)
* BUF_PTR1 (ptr to transmit buffer)
* I2C_ADDR2 (sub address)
*
* OUTPUT: I2C_ERROR byte (bit addressable)
*
* I2C_WRITE_MEMORY:
* MOV I2C_CTRL,#WRITE_MEMORY_MASK
* JMP SET_MESS_CNT
*
* I2C_WRITE_SUB_SWINC:
* MOV I2C_CTRL,#WRITE_SUB_SWINC_MASK
*
* SET_MESS_CNT:
* MOV MEM_MESS_LEN,I2C_MCB+1
* MOV I2C_MCB+1,#1 ;set BUF_LEN_1 = 1
*
* SUB_MESS:
* ACALL I2C_MESS_HAND
* JB I2C_ERR,END_WSWI
* INC I2C_MCB+2 ;inc. BUF_PTR_1
* INC I2C_MCB+3 ;inc. SUB_ADDR
* JNB RWN_BLK2,NEXT
* MOV MEM_DELAY_H,#EEPROM_PROG_DELAY
* MOV MEM_DELAY_L,#00 ; 40 ms delay at 16MHz
*
* PROGRAM_DELAY:
* DJNZ MEM_DELAY_L,$
* DJNZ MEM_DELAY_H,PROGRAM_DELAY
*
* NEXT:
* DJNZ MEM_MESS_LEN,SUB_MESS
*
* END_WSWI:
* RET
*
* END

```

EIE/AN91007

```
*TITLE (I2C_Write_Sub_Write command)
**
**
** SOURCE FILE : I2C_WSUW.ASM
** PACKAGE      : I2C
**
**
$DEBUG

**
** INCLUDES
**
$NCLIST
$IINCLUDE (I2C_DATA.GLO)
$LIST

**
** GLOBAL REFERENCES
**
EXTRN CODE(I2C_MESS_HAND)

**
** GLOBAL FUNCTION DEFINITIONS
**
PUBLIC I2C_WRITE_SUB_WRITE

**
** LOCAL SYMBOL DECLARATIONS
**
WRITE_SUB_WRITE MASK EQU 1CH
                ;REP_STRT_BLK1    = 0      (NO)
                ;RWN_BLK1         = 0      (WRITE)
                ;ADDR2            = 1      (YES)
                ;ADDR2 SUB       = 1      (YES)
                ;BLOCK2          = 1      (YES)
                ;RWN_BLK2        = 0      (WRITE)
                ;REP_STRT_BLK2   = 0      (NO)
                ;TEST_DEVICE     = 0      (--)

**
** CODE SEGMENT
**
I2C DRIVER SEGMENT CODE
RSEG I2C_DRIVER

**MPF...:I2C::I2C_WSUW.ASM:I2C_WRITE_SUB_WRITE
**
** FUNCTION NAME:      I2C_WRITE_SUB_WRITE
** PACKAGE:           I2C
** DESCRIPTION:
** Write 2 blocks of data (a and b, length n and m)
** preceded by a sub address into a single slave device
**
**
** PROTOCOL:
** <S><SLV ADDR><W><A><SUB_ADDR><A><Da0><A>..<A><Dan-1><A>
** <DB0><A>..<Dm-1><A><P>
**
** INPUT: Message control byte I2C_CTRL (bit addressable)
** Message control block I2C_MCB, containing:
```

[illegible]

¹²C driver routines for 8XC751/2 microcontrollers EIE/AN91007

```

**      Message control block I2C_MCB, containing:
**
**      I2C_ADDR1      (slave address)
**      BUF_LEN1       (number of bytes in block a)
**      PTR1           (ptr to block a)
**      I2C_ADDR2      (sub address)
**      BUF_LEN2       (number of bytes in block b)
**      PTR2           (ptr to block b)
**
**      OUTPUT: I2C_ERROR byte   (bit addressable)
**
**-----*EMP-----*
**
**_I2C_WRITE SUB_READ:
**      MOV      I2C_CTRL, @WRITE_SUB_READ_MASK
**      AJMP     I2C_MESS_HAND
**
**      END

```

EIE/AN91007

```

; I2C_ADDR1      (slave address first device)
; BUF_LEN1      (number of bytes in block a)
; BUF_PTR1      (ptr to block a)
; BUF_LEN1      (number of bytes in block b)
; BUF_PTR1      (ptr to block b)
;
; OUTPUT: I2C_ERROR byte  (bit addressable)
;
;-----
_I2C_WRITE_COM_WRITE:
    MOV     I2C_MCB+5,I2C_MCB+4
    MOV     I2C_MCB+4,I2C_MCB+3
    MOV     I2C_CTRL,#WRITE_COM_WRITE_MASK
    AJMP    I2C_MESS_HAND
END

```

¹²C driver routines for 8XC751/2 microcontrollers 250X8 101 25m EIE/AN91007

```

$TITLE(I2C_Write_Rep_Write command)
;
;*
;* SOURCE FILE : I2C_WREN.ASM
;* PACKAGE : I2C
;*
$DEBUG
;
;*
;* INCLUDES
;*
$NOLIST
$INCLUDE(I2C_DATA.GLO)
$LIST
;
;*
;* GLOBAL REFERENCES
;*
EXTRN CODE(I2C_MESS_HAND)
;
;*
;* GLOBAL FUNCTION DEFINITIONS
;*
PUBLIC I2C_WRITE_REP_WRITE
;
;*
;* LOCAL SYMBOL DECLARATIONS
;*
WRITE_REP_WRITE_MASK EQU 54H
;
; REP_STRT BLK1 = 0 (NO)
; RWN BLK1 = 0 (WRITE)
; ADDR2 = 1 (YES)
; ADDR2 SUB = 0 (NO)
; BLOCK2 = 1 (YES)
; RWN BLK2 = 0 (WRITE)
; REP_STRT BLK2 = 1 (YES)
; TEST_DEVICE = 0 (--)
;
;*
;* CODE SEGMENT
;*
I2C_DRIVER_SEGMENT CODE
RSEG I2C_DRIVER
;
;*
;* MF:::I2C::I2C_WREN.ASM:I2C_WRITE_REP_WRITE
;*
;* FUNCTION NAME: I2C_WRITE_REP_WRITE
;* PACKAGE: I2C
;* DESCRIPTION:
;* Write a block of data (a length n) to a slave device,
;* sent repeated start and write a block (b length m) to
;* another slave device.
;*
;*
;* PROTOCOL:
;* <S>(SLV ADDR1)<W><A><Da0><A><Da1>...<A><Dan-1><A>
;* <S>(SLV ADDR2)<W><A><Db0><A><Db1>...<A><Dbm-1><A><P>

```

```

* INPUT: Message control byte I2C_CTRL (bit addressable)
* Message control block I2C_MCB, containing:
*
* I2C_ADDR1 (slave address first device)
* BUF_LEN1 (number of bytes in block a)
* BUF_PTR1 (ptr to block a)
* I2C_ADDR2 (slave address second device)
* BUF_LEN1 (number of bytes in block b)
* BUF_PTR1 (ptr to block b)
*
* OUTPUT: I2C_ERROR byte (bit addressable)
*
*=====
_I2C_WRITE REP WRITE:
MOV I2C_CTRL,#WRITE_REP_WRITE_MASK
AJMP I2C_MESS_HAND

END

```

I2C driver routines for 8XC751/2 microcontrollers 8XC751/2 8XC752 EIE/AN91007

```

$TITLE(I2C_Write_Rep_Read command)
;
;-----*
; SOURCE FILE : I2C_WRER.ASM
; PACKAGE      : I2C
;-----*
$DEBUG
;
;-----*
; INCLUDES
;-----*
$NOLIST
$INCLUDE(I2C_DATA.GLO)
$LIST
;
;-----*
; GLOBAL REFERENCES
;-----*
EXTRN CODE(I2C_MESS_HAND)
;
;-----*
; GLOBAL FUNCTION DEFINITIONS
;-----*
PUBLIC I2C_WRITE_REP_READ
;
;-----*
; LOCAL SYMBOL DECLARATIONS
;-----*
WRITE_REP_READ_MASK EQU 74H
; REP STRT BLK1      = 0 (NO)
; RWN_BLK1           = 0 (WRITE)
; ADDR2              = 1 (YES)
; ADDR2 SUB          = 0 (NO)
; BLOCK2             = 1 (YES)
; RWN_BLK2           = 1 (READ)
; REP STRT BLK2      = 1 (YES)
; TEST_DEVICE        = 0 (--)
;
;-----*
; CODE SEGMENT
;-----*
I2C_DRIVER SEGMENT CODE
RSEG I2C_DRIVER
;
;MPF:::I2C::I2C_WRER.ASM:I2C_WRITE_REP_READ-----*
;
; FUNCTION NAME:      I2C_WRITE_REP_READ
; PACKAGE:           I2C
; DESCRIPTION:
; Write a block of data (a length n) to a slave device,
; sent repeated start and read a block (b length m) from
; another slave device.
;
; PROTOCOL:
; <S><SLV_ADDR1><W><A><Da0><A><Da1><A>..<A><Dan-1><A>
; <S><SLV_ADDR2><R><A><Db0><A><Db1><A>..<A><Dbm-1><N><P>
;

```

```

; * INPUT: Message control byte I2C_CTRL (bit addressable)
; * Message control block I2C_MCB, containing:
; * I2C_ADDR1 (slave address first device)
; * BUF_LEN1 (number of bytes in block a)
; * BUF_PTR1 (ptr to block a)
; * I2C_ADDR2 (slave address second device)
; * BUF_LEN1 (number of bytes in block b)
; * BUF_PTR1 (ptr to block b)
; * OUTPUT: I2C_ERROR byte (bit addressable)
; *
; *EMP-----*
;
I2C_WRITE_REP_READ:
MOV I2C_CTRL,#WRITE_REP_READ_MASK
AJMP I2C_MESS_HAND
END

```


I²C driver routines for 8XC751/2 microcontrollers EIE/AN91007

```

$TITLE(I2C_Read command)
;
;
; SOURCE FILE : I2C_READ.ASM
; PACKAGE : I2C
;
$DEBUG

;
; INCLUDES
;
$NOLIST
$INCLUDE(I2C_DATA.GLO)
$LIST

;
; GLOBAL REFERENCES
;
EXTRN CODE(I2C_MESS_HAND)

;
; GLOBAL FUNCTION DEFINITIONS
;
PUBLIC I2C_READ
PUBLIC _I2C_READ_STATUS

;
; LOCAL SYMBOL DECLARATIONS
;
READ_MASK EQU 02H
; REP_STRT_BLK1 = 0 (NO)
; RWN_BLK1 = 1 (READ)
; ADDR2 = 0 (NO)
; ADDR2 SUB = 0 (--)
; BLOCK2 = 0 (NO)
; RWN_BLK2 = 0 (--)
; REP_STRT_BLK2 = 0 (--)
; TEST_DEVICE = 0 (--)

;
; CODE SEGMENT
;
I2C DRIVER SEGMENT CODE
RSEG I2C_DRIVER

; *MPF:::I2C::I2C_READ.ASM:I2C_READ
;
; * FUNCTION NAME: I2C_READ
; * PACKAGE: I2C
; * DESCRIPTION:
; * Read a block of data from a slave device (READ) or read
; * a single byte from a slave device (READ STATUS)
; *
; * PROTOCOL:
; * <S><SLV_ADDR><R><A><D0><A><D1><A>...<A><Dn-1><N><P>
; * or
; * <S><SLV_ADDR><R><A><STATUS><N><P>
;

```

```

; * INPUT: Message control byte I2C_CTRL (bit addressable)
; * Message control block I2C_MCB, containing:
; * I2C_ADDR1 (slave address)
; * BUF_LEN1 (number of bytes in block)
; * BUF_PTR1 (ptr to store status)
; * OUTPUT: I2C_ERROR byte (bit addressable)
; *
; *EMP
;
_I2C_READ STATUS:
MOV I2C_MCB+2, I2C_MCB+1
MOV I2C_MCB+1, #1 ;buffer length = 1
_I2C_READ:
MOV I2C_CTRL, #READ_MASK
AJMP I2C_MESS_HAND

END

```

```

**TITLE(I2C_Read_Sub command)
**
**
**      SOURCE FILE : I2C_RSUB.ASM
**      PACKAGE      : I2C
**
**
**$DEBUG
**
**-----
**      INCLUDES
**-----
**$GLIST
**$INCLUDE(I2C_DATA.GLO)
**$LIST
**
**-----
**      GLOBAL REFERENCES
**-----
**      EXTRN CODE(I2C_MESS_HAND)
**
**-----
**      GLOBAL FUNCTION DEFINITIONS
**-----
**      PUBLIC I2C_READ_SUB
**
**-----
**      LOCAL SYMBOL DECLARATIONS
**-----
**      READ_SUB_MASK EQU 0FH
**
**      ;REP_STRT BLK1  = 1      (YES)
**      ;RWN_BLK1     = 1      (READ)
**      ;ADDR2        = 1      (YES)
**      ;ADDR2 SUB    = 1      (YES)
**      ;BLOCK2       = 0      (NO)
**      ;RWN_BLK2     = 0      (--)
**      ;REP_STRT BLK2 = 0      (--)
**      ;TEST_DEVICE  = 0      (--)
**
**-----
**      CODE SEGMENT
**-----
**
**      I2C DRIVER SEGMENT CODE
**      RSEG I2C_DRIVER
**
**-----
**$MPF:::I2C:::I2C_RSUB.ASM:I2C_READ_SUB
**
**      * FUNCTION NAME:      I2C_READ_SUB
**      * PACKAGE:           I2C
**      * DESCRIPTION:
**      *   Read a block of data (a length n) preceded by a
**      *   sub address from a slave device.
**      *
**      * PROTOCOL:
**      *   <S>SLV_ADDR<W><A><S>SUB_ADDR<A><S>SLV_ADDR<R><A>
**      *   <Da0><A>Da1<A>...<A>Da(n-1)<A><P>
**      *
**      * INPUT: Message control byte I2C_CTRL (bit addressable)
**      *   Message control block I2C_MCB, containing:

```

```

**      I2C_ADDR1      (slave address)
**      BUF_LEN1      (number of bytes in block )
**      BUF_PTR1      (ptr to block )
**      I2C_ADDR2      (sub address)
**
**      OUTPUT: I2C_ERROR byte  (bit addressable)
**
**-----*EMP
I2C_READ_SUB:
      MOV     I2C_CTRL,#READ_SUB_MASK
      AJMP    I2C_MESS_HAND
      END

```

EIE/AN91007

```

*MPF::I2C::I2C__RREW.ASM:I2C_READ_READ_WRITE
*
*
* * FUNCTION NAME:          I2C_READ_READ_WRITE
* * PACKAGE:              I2C
* * DESCRIPTION:
* *   Read a block of data (a length n) from a slave device,
* *   sent repeated start and write a block (b length m) to
* *   another slave device.
* *
* *
* * PROTOCOL:
* * <S><SLV_ADDRI><R><A><Da0><A><Da1><A>...<A><Dan-1><N>
* * <S><SLV_ADDR2><W><A><Db0><A><Db1><A>...<A><Dm-1><A><P>

```

12C driver routines for 8XC751/2 microcontrollers 8XC751/2 256Kbit E1E/AN91007

```

TITLE I2C_Read_Rep_Read command)
;
;*
;* SOURCE FILE : I2C_RRER.ASM
;* PACKAGE      : I2C
;*
;*=====
$DEBUG
;=====
;
;* INCLUDES
;*
;$NOLIST
$INCLUDE(I2C_DATA.GLO)
$LIST
;=====
; CODE SEGMENT
;=====
; GLOBAL REFERENCES
;=====
EXTRN CODE(I2C_MESS_HAND)
;=====
; GLOBAL FUNCTION DEFINITIONS
;=====
PUBLIC I2C_READ_REP_READ
;=====
; LOCAL SYMBOL DECLARATIONS
;=====
READ_REP_READ_MASK EQU 076H
;=====
; REP_STRT BLK1 = 0 (NO)
; RWN_BLK1 = 1 (READ)
; ADDR2 = 1 (YES)
; ADDR2 SUB = 0 (NO)
; BLOCKZ = 1 (YES)
; RWN_BLK2 = 1 (READ)
; REP_STRT BLK2 = 1 (YES)
; TEST_DEVICE = 0 (-- )
;=====
; CODE SEGMENT
;=====
I2C_READ_SEGMENT_CODE
RSEG I2C_DRIVER
;=====
;MPF:::I2C:::I2C_RRER.ASM:I2C_READ_REP_READ-----
;=====
;* FUNCTION NAME: I2C_READ_REP_READ
;* PACKAGE: I2C
;* DESCRIPTION:
;* Read a block of data (a length n) from a slave device,
;* sent repeated start and read a block (b length m) from
;* another slave device.
;*
;* PROTOCOL:
;* <S><SLV_ADDR2><R><A><Da0><A><Da1><A>...<A><Dan-1><N>
;* <S><SLV_ADDR2><R><A><Db0><A><Db1><A>...<A><Dm-1><N><P>

```

```

* INPUT: Message control byte I2C_CTRL (bit addressable)
* Message control block I2C_MCB, containing:
* I2C_ADDR1 (slave address first device)
* BUF_LEN1 (number of bytes in block a)
* BUF_PTR1 (ptr to block a)
* I2C_ADDR2 (slave address second device)
* BUF_LEN1 (number of bytes in block b)
* BUF_PTR1 (ptr to block b)
* OUTPUT: I2C_ERROR byte (bit addressable)
*-----*
*_I2C_READ_REP_READ:
    MOV     I2C_CTRL, #READ_REP_READ_MASK
    AJMP    I2C_MESS_HAND
    END

```

I²C driver routines for 8XC751/2 microcontrollers EIE/AN91007

I²C slave routines

I²C slave routines

I²C driver routines for 8XC751/2 microcontrollers 250X8 to 250X16 EIE/AN91007

```

$TITLE(I2C_SLAVE)
;
;
; SOURCE FILE : I2C_SLAV.ASM
; PACKAGE : I2C
;
; $DEBUG

;
; INCLUDES
;
$NOLIST
$INCLUDE(I2C_DATA.GLO)
$INCLUDE(REG751.H)
$LIST

;
; GLOBAL REFERENCES
;
;
EXTRN CODE(I2C_TRX_BYTE)
EXTRN CODE(I2C_RCV_BYTE)
EXTRN CODE(I2C_RCV_ADDR)
EXTRN CODE(ADDR_COMPARE)

;
; GLOBAL FUNCTION DEFINITIONS
;
PUBLIC I2C_SLV_TRX
PUBLIC I2C_SLV_RCV
PUBLIC ADDR_RECOG

;
; LOCAL SYMBOL DECLARATIONS
;
;
SLV_BUF_PTR_W SET R1
BIT_CNT SET R3
I2C_RELEASE EQU 0F4H

;
; *MPF:::I2C_SLAVE
;
; FUNCTION NAME: I2C_SLAVE_ROUTINES
; DESCRIPTION: I2C
; DESCRIPTION:
; This file contains an example of how to make a slave
; transmitter and slave receiver function. The slave
; transmitter functions transmits byte from a buffer,
; while the slave receiver routine receives bytes into
; a buffer. The buffer pointer is loaded during the
; I2C_INIT routine.
;
; *EMP

;
; INTERRUPT CODE SEGM: IIC
;
CSEG AT 023H
PUSH ACC
PUSH PSW
MOV PSW,#8

```

```

AJMP ADDR_RECOG

;
; CODE SEGMENT
;
; I2C DRIVER SEGMENT CODE
; RSEG I2C_DRIVER

;
; *MPF:::I2C:::I2C_SLAV.ASM:I2C_ADDR_RECOG
;
; FUNCTION NAME: I2C_ADDR_RECOG
; PACKAGE NAME: I2C
; DESCRIPTION:
; If an I2C interrupt occurs, and the STR bit is set,
; I2C_ADDR_RECOG receives the incoming slave address.
; If it's own slave address is recognized, the slave
; receiver or slave transmitter routine (depending on
; the R/WN bit) is called
;
; INPUT: --
; OUTPUT: I2C_ERROR byte (bit addressable)
;
; *EMP
ADDR_RECOG:
MOV I2CON,#C_STRT
ACALL I2C_RCV_ADDR
JB I2C_ERR_EXIT_SLV_AD
ACALL ADDR_COMPARE
EXIT_SLV_AD:
CLR I2C_ERR
MOV I2CON,#I2C_RELEASE
POP PSW
POP ACC
RETI

;
; *MPF:::I2C:::I2C_SLAV.ASM:I2C_SLV_TRX
;
; FUNCTION NAME: I2C_SLV_TRX
; PACKAGE NAME: I2C
; DESCRIPTION:
; After the SLV_ADDR/W is received, the I2C_SLV_TRX
; transmits a byte from the slave buffer. The pointer
; to this buffer is loaded during the I2C_INIT function.
; If an acknowledge is received, the pointer is
; incremented and the next byte is transmitted. The
; function is exit on reception of a NACK.
;
; Normally the slave routines are entered through an I2C
; interrupt, but if the 8x751 loses arbitration during
; the slave address and it recognises it's own slave
; address/W, the I2C_SLV_TRX function is entered at XXXX
;
; PROTOCOL: <S><SLV_ADDR><W><D0><A><D1><A>...<A><Dn><N><P>
;
; REGISTER USAGE : Register bank 1, is used during the I2C
; routines, it contains no static data, and is free
; for the user outside the I2C routines
;
; INPUT: --
; OUTPUT: I2C_ERROR byte (bit addressable)
;
; *EMP
I2C_SLV_TRX:

```


I²C driver routines for 8XC751/2 microcontrollers

```

S_TRX:    MOV     SLV_BUF_PTR_W,SLV_BUF_PTR
          MOV     A,8SLV_BUF_PTR_W
          ACALL   I2C_TRX_BYTE
          JB      I2C_ERR,EXIT_SLV_TRX
          JB      NO_ACK,EXIT_SLV_TRX
          INC     SLV_BUF_PTR_W
          AJMP    S_TRX
EXIT_SLV_TRX:
          RET

;MPF:::I2C::I2C_SLAV.ASM:I2C_SLV_RCV-----*
;*
;* FUNCTION NAME:      I2C_SLV_RCV
;* PACKAGE NAME:      I2C
;* DESCRIPTION:
;* After the SLV_ADDR/R is received, the I2C SLV RCV
;* receives a byte into the slave buffer. The pointer
;* to this buffer is loaded during the I2C INIT function.*
;* After the byte is received, an acknowledge is send,*
;* the pointer is incremented and the next byte is
;* received. The function is exit on if a start condition*
;* is detected
;*
;* Normally the slave routines are entered through an I2C*
;* interrupt, but if the 8x751 loses arbitration during
;* the slave address and it recognises it's own slave
;* address/R, the I2C_SLV_RCV function is entered at xx
;*
;* PROTOCOL: <S><SLV_ADDR><R><D0><A><D1><A>...<A><Dn><A><P>
;*
;* REGISTER USAGE : Register bank 1, is used during the I2C
;* routines, it contains no static data, and is free
;* for the user outside the I2C routines
;*
;* INPUT:  --
;* OUTPUT: I2C_ERROR byte (bit addressable)
;*
;*EMP-----*
I2C_SLV_RCV:
          MOV     SLV_BUF_PTR_W,SLV_BUF_PTR
S_RCV:    ACALL   I2C_RCV_BYTE
          JB      I2C_ERR,EXIT_SLV_RCV
          MOV     8SLV_BUF_PTR_W,A      ;save byte
          MOV     I2DAT,#0              ;send ACK
          JNB     AEN,$
          JNB     DRY,EXIT_SLV_RCV
          INC     SLV_BUF_PTR_W
          AJMP    S_RCV
EXIT_SLV_RCV:
          RET

;*-----*
;* H I S T O R Y
;*-----*
;* 21-05-91    J.C. Pijnenburg    original version
;*-----*
          END

```


I²C driver routines for 8XC751/2 microcontrollers

```

** DEFINE (I2C_INIT(Own_Slv_Addr,Slv_Buf_Addr,Retry))
{
    MOV OWN_SLV_ADDR,%%Own_Slv_Addr
    MOV SLV_BUF_PTR, %%Slv_Buf_Addr
    MOV I2C_MCB, %%RetFy
    ACALL _I2C_INIT
}

** DEFINE (I2C_TEST_DEVICE(Slv_Addr))
{
    MOV I2C_MCB,%%Slv_Addr
    ACALL _I2C_TEST_DEVICE
}

** DEFINE (I2C_WRITE(Slv_Addr,Count,Source_Ptr))
{
    MOV I2C_MCB, %%Slv_Addr
    MOV I2C_MCB+1,%%Count
    MOV I2C_MCB+2,%%Source_Ptr
    ACALL _I2C_WRITE
}

** DEFINE (I2C_WRITE_SUB(Slv_Addr,Count,Source_Ptr,Sub_Addr))
{
    MOV I2C_MCB, %%Slv_Addr
    MOV I2C_MCB+1,%%Count
    MOV I2C_MCB+2,%%Source_Ptr
    MOV I2C_MCB+3,%%Sub_Addr
    ACALL _I2C_WRITE_SUB
}

** DEFINE (I2C_WRITE_SUB_SWINC(Slv_Addr,Count,Source_Ptr,Sub_Addr))
{
    MOV I2C_MCB, %%Slv_Addr
    MOV I2C_MCB+1,%%Count
    MOV I2C_MCB+2,%%Source_Ptr
    MOV I2C_MCB+3,%%Sub_Addr
    ACALL _I2C_WRITE_SUB_SWINC
}

** DEFINE (I2C_WRITE_MEMORY(Slv_Addr,Count,Source_Ptr,Sub_Addr))
{
    MOV I2C_MCB, %%Slv_Addr
    MOV I2C_MCB+1,%%Count
    MOV I2C_MCB+2,%%Source_Ptr
    MOV I2C_MCB+3,%%Sub_Addr
    ACALL _I2C_WRITE_MEMORY
}

** DEFINE (I2C_WRITE_SUB_WRITE(Slv_Addr,Count_1,Source_Ptr_1,Sub_Addr,Count_2,Source_Ptr_2))
{
    MOV I2C_MCB, %%Slv_Addr
    MOV I2C_MCB+1,%%Count_1
    MOV I2C_MCB+2,%%Source_Ptr_1
    MOV I2C_MCB+3,%%Sub_Addr
    MOV I2C_MCB+4,%%Count_2
    MOV I2C_MCB+5,%%Source_Ptr_2
    ACALL _I2C_WRITE_SUB_WRITE
}

** DEFINE (I2C_WRITE_SUB_READ(Slv_Addr,Count_1,Source_Ptr,Sub_Addr,Count_2,Dest_Ptr))
{
    MOV I2C_MCB, %%Slv_Addr
    MOV I2C_MCB+1,%%Count_1
    MOV I2C_MCB+2,%%Source_Ptr
    MOV I2C_MCB+3,%%Sub_Addr
    MOV I2C_MCB+4,%%Count_2
    MOV I2C_MCB+5,%%Dest_Ptr
    ACALL _I2C_WRITE_SUB_READ
}

MOV I2C_MCB+2,%%Source_Ptr
MOV I2C_MCB+3,%%Sub_Addr
MOV I2C_MCB+4,%%Count_2
MOV I2C_MCB+5,%%Dest_Ptr
ACALL _I2C_WRITE_SUB_READ

** DEFINE (I2C_WRITE_COM_WRITE(Slv_Addr,Count_1,Source_Ptr_1,Count_2,Source_Ptr_2))
{
    MOV I2C_MCB, %%Slv_Addr
    MOV I2C_MCB+1,%%Count_1
    MOV I2C_MCB+2,%%Source_Ptr_1
    MOV I2C_MCB+3,%%Count_2
    MOV I2C_MCB+4,%%Source_Ptr_2
    ACALL _I2C_WRITE_COM_WRITE
}

** DEFINE (I2C_WRITE_REP_WRITE(Slv_Addr,Count_1,Source_Ptr_1,Sub_Addr,Count_2,Source_Ptr_2))
{
    MOV I2C_MCB, %%Slv_Addr
    MOV I2C_MCB+1,%%Count_1
    MOV I2C_MCB+2,%%Source_Ptr_1
    MOV I2C_MCB+3,%%Sub_Addr
    MOV I2C_MCB+4,%%Count_2
    MOV I2C_MCB+5,%%Source_Ptr_2
    ACALL _I2C_WRITE_REP_WRITE
}

** DEFINE (I2C_WRITE_REP_READ(Slv_Addr,Count_1,Source_Ptr,Sub_Addr,Count_2,Dest_Ptr))
{
    MOV I2C_MCB, %%Slv_Addr
    MOV I2C_MCB+1,%%Count_1
    MOV I2C_MCB+2,%%Source_Ptr
    MOV I2C_MCB+3,%%Sub_Addr
    MOV I2C_MCB+4,%%Count_2
    MOV I2C_MCB+5,%%Dest_Ptr
    ACALL _I2C_WRITE_REP_READ
}

** DEFINE (I2C_READ(Slv_Addr,Count,Dest_Ptr))
{
    MOV I2C_MCB, %%Slv_Addr
    MOV I2C_MCB+1,%%Count
    MOV I2C_MCB+2,%%Dest_Ptr
    ACALL _I2C_READ
}

** DEFINE (I2C_READ_STATUS(Slv_Addr,Dest_Ptr))
{
    MOV I2C_MCB, %%Slv_Addr
    MOV I2C_MCB+1,%%Dest_Ptr
    ACALL _I2C_READ_STATUS
}

** DEFINE (I2C_READ_SUB(Slv_Addr,Count,Dest_Ptr,Sub_Addr))
{
    MOV I2C_MCB, %%Slv_Addr
    MOV I2C_MCB+1,%%Count
    MOV I2C_MCB+2,%%Dest_Ptr
    MOV I2C_MCB+3,%%Sub_Addr
    ACALL _I2C_READ_SUB
}

```

EIE/AN91007

```

1) DEFINE (I2C_READ_REP_READ (Slv_Addr, Count_1, Dest_Ptr, Sub_Addr, Count_2, Source_Ptr_2))
{
    MOV I2C MCB, ##Slv_Addr
    MOV I2C MCB+1, ##Count_1
    MOV I2C MCB+2, ##Dest_Ptr_1
    MOV I2C MCB+3, ##Sub_Addr
    MOV I2C MCB+4, ##Count_2
    MOV I2C MCB+5, ##Dest_Ptr_2
    ACALL I2C_READ_REP_READ
}

2) DEFINE (I2C_READ_REP_WRITE (Slv_Addr, Count_1, Dest_Ptr, Sub_Addr, Count_2, Source_Ptr))
{
    MOV I2C MCB, ##Slv_Addr
    MOV I2C MCB+1, ##Count_1
    MOV I2C MCB+2, ##Dest_Ptr
    MOV I2C MCB+3, ##Sub_Addr
    MOV I2C MCB+4, ##Count_2
    MOV I2C MCB+5, ##Source_Ptr
    ACALL I2C_READ_REP_WRITE
}

3) DEFINE (I2C_WRITE_SINGLE (Slv_Addr, Dest_Ptr, Sub_Addr, Count_1, Source_Ptr))
{
    MOV I2C MCB, ##Slv_Addr
    MOV I2C MCB+1, ##Dest_Ptr
    MOV I2C MCB+2, ##Sub_Addr
    MOV I2C MCB+3, ##Count_1
    MOV I2C MCB+4, ##Source_Ptr
    ACALL I2C_WRITE_SINGLE
}

4) DEFINE (I2C_READ_SINGLE (Slv_Addr, Dest_Ptr, Sub_Addr, Count_1, Source_Ptr))
{
    MOV I2C MCB, ##Slv_Addr
    MOV I2C MCB+1, ##Dest_Ptr
    MOV I2C MCB+2, ##Sub_Addr
    MOV I2C MCB+3, ##Count_1
    MOV I2C MCB+4, ##Source_Ptr
    ACALL I2C_READ_SINGLE
}

5) DEFINE (I2C_WRITE_MULTIPLE (Slv_Addr, Dest_Ptr, Sub_Addr, Count_1, Source_Ptr))
{
    MOV I2C MCB, ##Slv_Addr
    MOV I2C MCB+1, ##Dest_Ptr
    MOV I2C MCB+2, ##Sub_Addr
    MOV I2C MCB+3, ##Count_1
    MOV I2C MCB+4, ##Source_Ptr
    ACALL I2C_WRITE_MULTIPLE
}

6) DEFINE (I2C_READ_MULTIPLE (Slv_Addr, Dest_Ptr, Sub_Addr, Count_1, Source_Ptr))
{
    MOV I2C MCB, ##Slv_Addr
    MOV I2C MCB+1, ##Dest_Ptr
    MOV I2C MCB+2, ##Sub_Addr
    MOV I2C MCB+3, ##Count_1
    MOV I2C MCB+4, ##Source_Ptr
    ACALL I2C_READ_MULTIPLE
}

```

I²C driver routines for 8XC751/2 microcontrollers EIE/AN91007

I2C_PLM.H

I2C driver routines for 8XC751/2 microcontrollers

```
/*  
/*  
/* INCLUDE FILE: I2C.PLM.H  
/* PACKAGE: I2C  
/*  
/*  
/*  
/*  
/*  
/*  
/* GLOBAL FUNCTION PROTOTYPES  
/*  
/*  
I2C_INIT: PROCEDURE(Own_Slv_Addr,Slv_Buf_Addr,Retry) BIT EXTERNAL;  
  DECLARE(Own_Slv_Addr,Slv_Buf_Addr,Retry) BYTE MAIN;  
END I2C_INIT;  
  
I2C_TEST_DEVICE: PROCEDURE(Slv_Addr) BIT EXTERNAL;  
  DECLARE(Slv_Addr) BYTE MAIN;  
END I2C_TEST_DEVICE;  
  
I2C_WRITE: PROCEDURE(Slv_Addr,Count,Source_Ptr) BIT EXTERNAL;  
  DECLARE(Slv_Addr,Count,Source_Ptr) BYTE MAIN;  
END I2C_WRITE;  
  
I2C_WRITE_SUB: PROCEDURE(Slv_Addr,Count,Source_Ptr,Sub_Addr) BIT EXTERNAL;  
  DECLARE(Slv_Addr,Count,Source_Ptr,Sub_Addr) BYTE MAIN;  
END I2C_WRITE_SUB;  
  
I2C_WRITE_SUB_SWINC: PROCEDURE(Slv_Addr,Count,Source_Ptr,Sub_Addr) BIT EXTERNAL;  
  DECLARE(Slv_Addr,Count,Source_Ptr,Sub_Addr) BYTE MAIN;  
END I2C_WRITE_SUB_SWINC;  
  
I2C_WRITE_MEMORY: PROCEDURE(Slv_Addr,Count,Source_Ptr,Sub_Addr) BIT EXTERNAL;  
  DECLARE(Slv_Addr,Count,Source_Ptr,Sub_Addr) BYTE MAIN;  
END I2C_WRITE_MEMORY;  
  
I2C_WRITE_SUB_WRITE: PROCEDURE(Slv_Addr,Count_1,Source_Ptr_1,Sub_Addr,Count_2,Source_Ptr_2) BIT EXTERNAL;  
  DECLARE(Slv_Addr,Count_1,Source_Ptr_1,Sub_Addr,Count_2,Source_Ptr_2) BYTE MAIN;  
END I2C_WRITE_SUB_WRITE;  
  
I2C_WRITE_SUB_READ: PROCEDURE(Slv_Addr,Count_1,Source_Ptr,Sub_Addr,Count,Dest_Ptr) BIT EXTERNAL;  
  DECLARE(Slv_Addr,Count_1,Source_Ptr,Sub_Addr,Count,Dest_Ptr) BYTE MAIN;  
END I2C_WRITE_SUB_READ;  
  
I2C_WRITE_COM_WRITE: PROCEDURE(Slv_Addr,Count_1,Source_Ptr_1,Count_2,Source_Ptr_2) BIT EXTERNAL;  
  DECLARE(Slv_Addr,Count_1,Source_Ptr_1,Count_2,Source_Ptr_2) BYTE MAIN;  
END I2C_WRITE_COM_WRITE;  
  
I2C_WRITE_REP_WRITE: PROCEDURE(Slv_Addr,Count_1,Source_Ptr_1,Sub_Addr,Count_2,Source_Ptr_2) BIT EXTERNAL;  
  DECLARE(Slv_Addr,Count_1,Source_Ptr_1,Sub_Addr,Count_2,Source_Ptr_2) BYTE MAIN;  
END I2C_WRITE_REP_WRITE;  
  
I2C_WRITE_REP_READ: PROCEDURE(Slv_Addr,Count_1,Source_Ptr,Sub_Addr,Count_2,Dest_Ptr) BIT EXTERNAL;  
  DECLARE(Slv_Addr,Count_1,Source_Ptr,Sub_Addr,Count_2,Dest_Ptr) BYTE MAIN;  
END I2C_WRITE_REP_READ;  
  
I2C_READ: PROCEDURE(Slv_Addr,Count,Dest_Ptr) BIT EXTERNAL;  
  DECLARE(Slv_Addr,Count,Dest_Ptr) BYTE MAIN;  
END I2C_READ;
```

```
I2C_READ_STATUS: PROCEDURE(Slv_Addr,Dest_Ptr) BIT EXTERNAL;  
  DECLARE(Slv_Addr,Dest_Ptr) BYTE MAIN;  
END I2C_READ_STATUS;  
  
I2C_READ_SUB: PROCEDURE(Slv_Addr,Count,Dest_Ptr,Sub_Addr) BIT EXTERNAL;  
  DECLARE(Slv_Addr,Count,Dest_Ptr,Sub_Addr) BYTE MAIN;  
END I2C_READ_SUB;  
  
I2C_READ_REP_READ: PROCEDURE(Slv_Addr,Count_1,Dest_Ptr_1,Sub_Addr,Count_2,Dest_Ptr_2) BIT EXTERNAL;  
  DECLARE(Slv_Addr,Count_1,Dest_Ptr_1,Sub_Addr,Count_2,Dest_Ptr_2) BYTE MAIN;  
END I2C_READ_REP_READ;  
  
I2C_READ_REP_WRITE: PROCEDURE(Slv_Addr,Count_1,Dest_Ptr_1,Sub_Addr,Count_2,Source_Ptr) BIT EXTERNAL;  
  DECLARE(Slv_Addr,Count_1,Dest_Ptr_1,Sub_Addr,Count_2,Source_Ptr) BYTE MAIN;  
END I2C_READ_REP_WRITE;
```

H.M.I. 351

I²C driver routines for 8XC751/2 microcontrollers EIE/AN91007

I²C driver routines for 8XC751/2 microcontrollers EIE/AN91007

```

/*=====*/
/*
/*      INCLUDE FILE:   I2C_C.H
/*      PACKAGE:       I2C
/*
/*=====*/

/*=====*/
/*      GLOBAL FUNCTION PROTOTYPES
/*=====*/

bit I2C_INIT(char Own_Slv_Addr, char *Slv_Buf_Ptr, char Retry);
bit I2C_TEST_Device(char Slv_Addr);
bit I2C_WRITE(char Slv_Addr, char Count, char *Source_Ptr);
bit I2C_WRITE_SUB(char Slv_Addr, char Count, char *Source_Ptr, char Sub_Addr);
bit I2C_WRITE_SUB_SWINC(char Slv_Addr, char Count, char *Source_Ptr, char Sub_Addr);
bit I2C_WRITE_MEMORY(char Slv_Addr, char Count, char *Source_Ptr, char Sub_Addr);
bit I2C_WRITE_SUB_WRITE(char Slv_Addr, char Count_1, char *Source_Ptr_1, char Sub_Addr, char Count_2,
char *Source_Ptr_2);
bit I2C_WRITE_SUB_READ(char Slv_Addr, char Count_1, char *Source_Ptr, char Sub_Addr, char Count, char
*Dest_Ptr);
bit I2C_WRITE_COM_WRITE(char Slv_Addr, char Count_1, char *Source_Ptr_1, char Count_2, char *Source_
Ptr_2);
bit I2C_WRITE_REP_WRITE(char Slv_Addr, char Count_1, char *Source_Ptr_1, char Sub_Addr, char Count_2
, char *Source_Ptr_2);
bit I2C_WRITE_REP_READ(char Slv_Addr, char Count_1, char *Source_Ptr, char Sub_Addr, char Count_2, ch
ar *Dest_Ptr);
bit I2C_READ(char Slv_Addr, char Count, char *Dest_Ptr);
bit I2C_READ_STATUS(char Slv_Addr, char *Dest_Ptr);
bit I2C_READ_SUB(char Slv_Addr, char Count, char *Dest_Ptr, char Sub_Addr);
bit I2C_READ_REP_READ(char Slv_Addr, char Count_1, char *Dest_Ptr_1, char Sub_Addr, char Count_2, ch
ar *Dest_Ptr_2);
bit I2C_READ_REP_WRITE(char Slv_Addr, char Count_1, char *Dest_Ptr_1, char Sub_Addr, char Count_2, ch
ar *Source_Ptr);

```

I²C driver routines for 8XC751/2 microcontrollers EIE/AN91007

microcontrollers 16C0X8 7

```

$CASE
$TITLE(Assembly example program)
;-----
;*
;* SOURCE FILE : DEMO ASM.ASM
;* PACKAGE : I2C
;*
;* Hours and minutes will be displayed on LCD display
;* Dot between hours and minutes will blink
;*-----
$DEBUG
$NOLIST

;-----
;* INCLUDES
;*-----
$INCLUDE(I2C_DATA.GLO)
$INCLUDE(I2C_MAC.DEF)
$LIST

;-----
;* GLOBAL FUNCTION DEFINITIONS
;*-----
EXTRN CODE( I2C_INIT)
EXTRN CODE( I2C_WRITE)
EXTRN CODE( I2C_READ_SUB)

;-----
;* LOCAL DATA DEFINITIONS
;*-----
RAMVAR SEGMENT DATA ;Segment for variables
STACK SEGMENT DATA ;Segment for variables
USER SEGMENT CODE ;Segment for application program

;-----
;* LOCAL SYMBOL DEFINITIONS
;*-----
CLOCK_ADR EQU 0A2H ;Address of PCF8583
CL_SUB_ADR EQU 01H ;Sub address for reading time
LCD_ADR EQU 74H ;Address of PCF8577

;-----
;* DATA SEGMENT
;*-----
RSEG RAMVAR
TIME_BUFFER: DS 4 ;Buffer for I2C strings
LCD_BUFFER: DS 5

RSEG STACK
STACK_DATA: DS 10

;-----
;* CODE SEGMENT
;*-----
CSEG AT 00
AJMP APPL_MAIN

RSEG USER

```

```

APPL_MAIN:
    MOV SP,$STACK_DATA-1
    MOV DPTR,#LCD_TAB ;Pointer to segment table
    MOV LCD_BUFFER,$00 ;Ctrl word for LCD driver
    $12C_INIT($2h,TIME_BUFFER,0) ;Init I2C interface
    CLR A ;Prepare buffer
    MOV TIME_BUFFER,A ;for clock int.
    MOV TIME_BUFFER+1,A
    $12C_WRITE(CLOCK_ADR,2,TIME_BUFFER) ;Initialise clock

REPEAT: $12C_READ_SUB(CLOCK_ADR,4,TIME_BUFFER,CL_SUB_ADR)
        ;Read time

;-----*
;* Time has been read. Order: *
;* hundreds of sec's, sec's, min's and hr's *
;*-----*
MOV A,TIME_BUFFER+3 ;Mask of hour counter
ANL A,#3FH
MOV TIME_BUFFER+3,A
ACALL CONVERT ;Convert time data to
                ;LCD segment data

;-----*
;* Check if dot has to be switched on *
;*-----*
ORL LCD_BUFFER+3,$01h

;-----*
;* If lab of seconds is '0', then switch on dp *
;*-----*
MOV A,TIME_BUFFER+1 ;Get seconds
RRC A
JC PROCEED
ORL LCD_BUFFER+1,$01 ;Switch on dp

;-----*
;* Display new time *
;*-----*
PROCEED: $12C_WRITE(LCD_ADR,5,LCD_BUFFER) ;Read new time
        SMPF REPEAT

;-----*
;* CONVERT converts BCD data of time to segment data *
;*-----*
CONVERT: MOV R0,#LCD_BUFFER+1 ;R0 is pointer
        MOV A,TIME_BUFFER+3 ;Get hours
        SWAP A ;Swap nibbles
        ACALL LCD_DATA ;Convert 10's of hours
        MOV A,TIME_BUFFER+3
        ACALL LCD_DATA ;Convert hours
        MOV A,TIME_BUFFER+2 ;Get minutes
        SWAP A
        ACALL LCD_DATA ;Convrt 10's of minut
        MOV A,TIME_BUFFER+2
        ACALL LCD_DATA ;Convert minutes
        RET

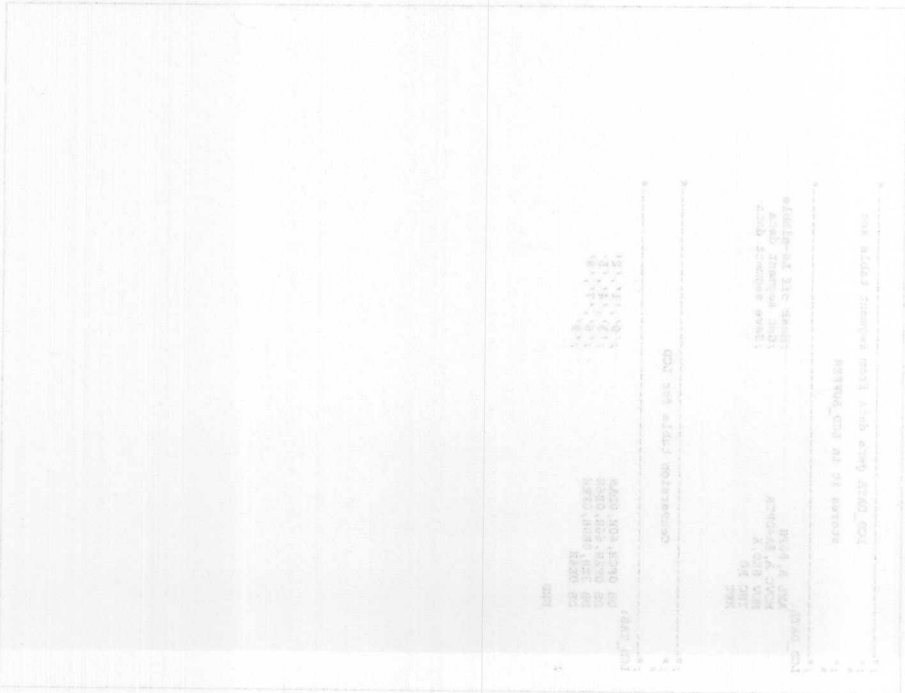
```

I²C driver routines for 8XC751/2 microcontrollers 25CXX101 25CXX102 EIE/AN91007

```
;*-----*  
;*      LCD_DATA gets data from segment table and  
;*      stores it in LCD_BUFFER  
;*-----*  
LCD_DATA: ANL A,#0FH           ;Mask off LS-nibble  
          MOVC A,@A+DPTR       ;Get segment data  
          MOV @R0,A           ;Save segment data  
          INC R0  
          RET  
  
;*-----*  
;*      Conversion table for LCD  
;*-----*  
LCD_TAB: DB 0FCH,60H,0DAH      ;'0','1','2'  
          DB 0F2H,66H,0B6H      ;'3','4','5'  
          DB 3EH,0E0H,0FEH      ;'6','7','8'  
          DB 0E6H               ;'9'  
          ;  
          END
```

I²C driver routines for 8XC751/2 microcontrollers EIE/AN91007

DEMO_PLM.PLM



I²C driver routines for 8XC751/2 microcontrollers

```

Tab_Point=.LCD_Tab+(Time AND 0FH); /* MIN's */
Segment=Tab_Value;
Time_Point=.Time_Buffer(1); /* Check sec's for
                               blinking */

LCD_Point=.LCD_Buffer+1;
If (Time AND 01H)>0 then
    Segment=(Segment OR 01H);
Call I2C_WRITE(LCD_Adr,5,.LCD_Buffer);
                               /* Display time */

End;

End Demo;

End Demo_plm;

```

```

$OPTIMIZE(4)
$DEBUG
$CODE
/*=====*/
/*
SOURCE FILE : DEMO_PLM.PLM
PACKAGE      : I2C
*/
/* Hours and minutes will be displayed on LCD display */
/* Dot between hours and minutes will blink */
/*=====*/

Demo_plm: Do;
/*=====*/
/* INCLUDES */
/*=====*/
$NOLIST
$INCLUDE(I2C_PLM.H)
$LIST

/*=====*/
/* MAIN */
/*=====*/

Clock: Do;
/* Variable and constant declarations */

Declare LCD_TAB(*) Byte Constant (0Fh,60H,0DAh,
                                0F2H,66H,0B6H,3EH,0E0H,0FEH,0E6H);
Declare Time_Buffer(4) Byte Main;
Declare LCD_Buffer(5) Byte Main;
Declare Tab_Point Word Main;
Declare (LCD_Point,Time_Point) Byte Main;
Declare Segment Based LCD_Point Byte Main;
Declare Time Based Time_Point Byte Main;
Declare Tab_Value Based Tab_Point Byte Constant;

Declare Clock_Adr Literally '0A2h';
Declare LCD_Adr Literally '74h';
Declare Cl_Sub_Adr Literally '01h';

Call I2C_INIT(22h,.Time_Buffer,0);
LCD_Buffer(0)=0; /* LCD control word */
Time_Buffer(0)=0;
Time_Buffer(1)=0;
Call I2C_WRITE(Clock_Adr,2,.Time_Buffer);
/* Initialise clock */

Do While LCD_Buffer(0)=0; /* Program loop */
    Call I2C_READ_SUB(Clock_Adr,4,.Time_Buffer,
                    Cl_Sub_Adr); /* Get time */
    LCD_Point=.LCD_Buffer(1); /* Init pointers */
    Time_Point=.Time_Buffer(3);
    Tab_Point=.LCD_Tab(0)+SHR(Time,4); /* 10-HR's */
    Segment=Tab_Value;
    LCD_Point=LCD_Point+1;
    Tab_Point=.LCD_Tab(0)+(Time AND 0FH); /* HR's */
    Segment=Tab_Value;
    Time_Point=Time_Point-1;
    LCD_Point=LCD_Point+1;
    Tab_Point=.LCD_Tab+SHR(Time,4); /* 10-MIN's */
    Segment=(Tab_Value OR 01H); /* dp */
    LCD_Point=LCD_Point+1;

```

DEMO_C.C

I2C driver routines for 8XC751/2 microcontrollers EIE/AN91007

DEMO_C.C

I²C driver routines for 8XC751/2 microcontrollers

EIE/AN91007

```

/*-----*/
/*
/* SOURCE FILE : DEMO.C.C
/* PACKAGE : I2C_DEMO
/*-----*/
/*MPP:::XXXXXX*/
/*
/* PACKAGE NAME: I2C_DEMO
/* DESCRIPTION:
/* This demo program reads the time from a PCF8583 clock*/
/* IC, and displays it to an LCD display (both available*/
/* at the I2C demoboard.
/*
/* Hours and minutes will be displayed on LCD display
/* Dot between hours and minutes will blink
/*
/*EMP*/
/*-----*/
/*
/* INCLUDES
/*-----*/
#include "I2C_C.H"
/*-----*/
/*
/* LOCAL SYMBOL DECLARATIONS
/*-----*/
#define Clock_Adr 0xA2
#define LCD_Adr 0x74
#define Cl_Sub_Adr 0x01
/*-----*/
/*
/* LOCAL DATA DEFINITIONS
/*-----*/
rom char LCD_Tab[]={0xFC,0x60,0xDA,0xF2,0x66,
0xB6,0x3E,0xE0,0xFE,0xE6};
/*-----*/
/*
/* MAIN
/*-----*/
void main()
{
rom char *Tab_Ptr;
data char Time_Buffer[4];
data char *Time_Ptr;
data char LCD_Buffer[5];
data char *LCD_Ptr;

I2C_INIT(0x22,&Time_Buffer,0);
LCD_Buffer[0]=0; /* LCD control word*/
Time_Buffer[0]=0;
Time_Buffer[1]=0;
I2C_WRITE(Clock_Adr,2,&Time_Buffer); /* Init clock*/

while (1) /* Program loop*/
{
I2C_READ_SUB(Clock_Adr,4,&Time_Buffer,Cl_Sub_Adr);
LCD_Ptr = &LCD_Buffer[1]; /* Get time*/
Time_Ptr = &Time_Buffer[3]; /* Init pointers*/
Tab_Ptr = (LCD_Tab+(*Time_Ptr >> 4)); /*10-HR's*/
*(LCD_Ptr++) =*Tab_Ptr;
Tab_Ptr = (LCD_Tab+(*Time_Ptr-- & 0x0F)); /* HR's*/
*(LCD_Ptr++) =*Tab_Ptr;
Tab_Ptr = (LCD_Tab+(*Time_Ptr >> 4)); /* 10-MIN's*/
*(LCD_Ptr++) = (*Tab_Ptr | 0x01); /* dp*/
Tab_Ptr = (LCD_Tab+(*Time_Ptr & 0x0F)); /* MIN's*/
*LCD_Ptr =*Tab_Ptr;
Time_Ptr = &Time_Buffer[1]; /* Check sec's blinking*/
LCD_Ptr = &LCD_Buffer[1];
if ((*Time_Ptr & 0x01)>0)
*LCD_Ptr = (*LCD_Ptr | 0x01);
I2C_WRITE(LCD_Adr,5,&LCD_Buffer); /* Display time*/
}
}

```

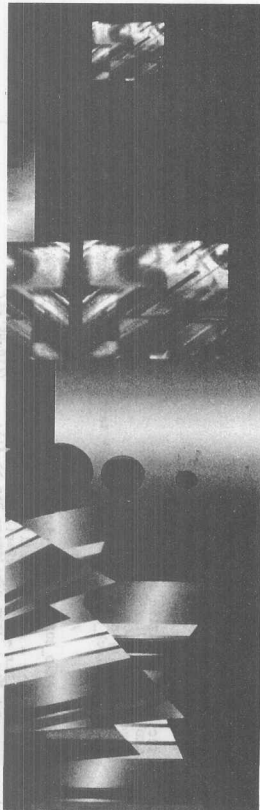
Exploring I²C

Authors: Steven Sarns and Jack Woehr

Embedded Systems™

PROGRAMMING

Exploring I²C



Serial data buses are a well-proven tool in embedded systems. When you are communicating with slow peripheral devices, serial buses are often more convenient and less expensive than parallel buses. Additionally, a serial interface featuring a UART or similar intermediary chip can also serve to isolate the CPU from noise and line glitches that might bring down the house if they were to occur on the processor bus. Peripherals can usually be controlled over a much greater distance by a serial bus. The serial approach offers greater resilience and noise immunity.

The price you pay for the convenience is a slower transmission rate and, possibly, the need for added interface circuitry at higher voltages. Many peripheral devices, however, are not in constant communication with the CPU and are not greatly affected by a slower bus. On the hardware side, any added interface circuitry required for serial-bus support is frequently compensated for by the resulting simplicity and tighter pinout of the serial peripherals.

CHOOSING THE PROPER ROUTE

Having decided that a serial bus makes sense for your application, your next task is to select the most appropriate bus and protocol. Here, as with rapid transit, your choice should be determined by your destination. Contrary to what some people may tell you, the choice of bus and protocol depends at least as much on the nature of the system's software as it does on the manufacturer's data sheets.

Consider, for example, the serial-peripheral interface (SPI) and multidrop

The choice of bus and protocol depends at least as much on the system's software as it does on the manufacturer's data sheets.

serial buses. Both buses are popular, but each exhibits severely constrained performance in large networks. SPI, as embodied in the Motorola 6800 family, was designed primarily for one-on-one exchanges between two devices. Similarly, the multidrop approach used in various 8051 family members as well as in the 68HC11 and various UART chips finds its broadest expression in RS485/422 half-duplex transmissions. Multidrop has no deterministic arbitration scheme between multiple masters, leaving it mainly suitable for single-master multiple-slave situations. (For more on multidrop, see Jack Woehr's article, "Multidrop Processing," *Embedded Systems Programming*, March 1990, pp 58-67—ed.) A different approach is to use a three-wire protocol called MicroWire, available from National Semiconductor in Santa Clara, Calif., which is fine for use with addressable peripherals, but requires an individual chip select for each device ad-

Exploring I²CExploring I²C

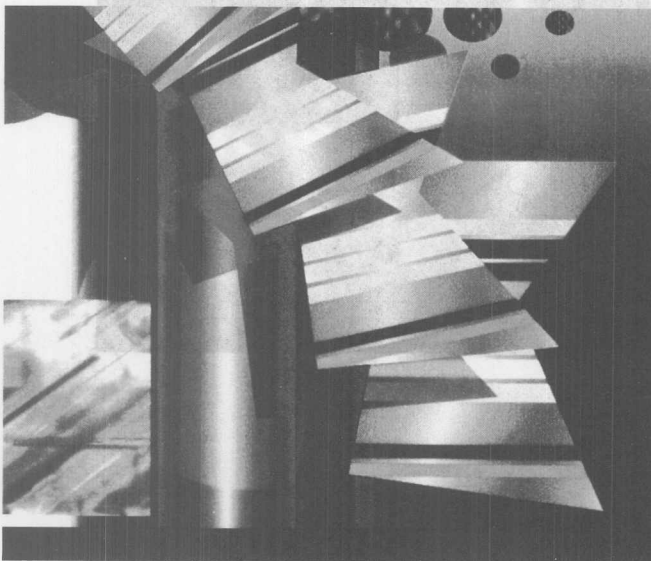
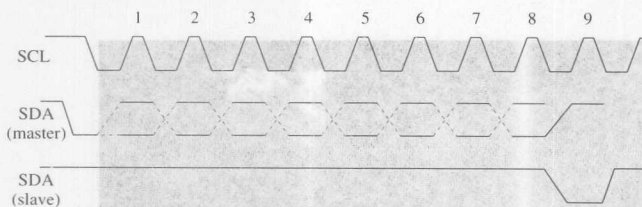
dressed. The added wiring offers no advantage to developers, and the bus offers nothing towards achieving multiple-mastering capabilities.

One of the more versatile options available to developers is the I²C bus promulgated by Philips/Signetics in Sunnyvale, Calif. I²C allows you to set

up a multiple-master, multiple-slave communications bus with conflict arbitration, using only twisted-pair wiring to connect the processors and peripherals. Philips/Signetics has moved to support this protocol (which is quite popular in Europe) with a large assortment of interesting doodads, and is actively

Figure 1

Generation of acknowledge.



Open-collector configuration means that the output stage can only pull the node to ground.

encouraging other manufacturers to join in the fun. If your next design features a microprocessor that supports I²C or you are prepared to implement I²C in software using a PIA as this article illustrates, your reward could be a decreased chip count and lower power consumption—along with a comfortable distributed-programming model for peripheral devices.

I²C is more flexible than the protocols noted above, since only two wires are required to service a large network of addressable masters and addressable slaves. A third wire may be added if interrupt service is required, though Philips/Signetics microprocessors featuring I²C support feature on-chip circuitry and are capable of interrupting the processor upon receipt of a valid address.

HOW I²C WORKS

The I²C bus consists of two lines: serial clock (SCL) and serial data (SDA). The beauty of the I²C bus is that each of these lines is bidirectional. Bidirectional means that everything on the bus is equal, unlike most other serial-peripheral busses such as SPI or MicroWire, which have dedicated inputs and outputs. Each I²C transaction line (SCL and SDA) is an open collector of output and input. The

Exploring I²CExploring I²C

pullup resistor is external.

Open-collector (actually, they are CMOS, so "open drain" is more appropriate) configuration means that the output stage can only pull the node to ground. A passive resistor pulls the node high, which means that any number of open collector outputs can be connected together with no deleterious results, because it is impossible to pull more current through the resistor than any one output will produce. Tying outputs together will produce disastrous results if the same procedure is tried with standard TTL outputs. If some of the outputs go high and some are low, the current is unlimited and the logic level of the output will be in an indeterminate state. Tying open-collector outputs together is also known as "wire ORing" because if either A or B goes low, so does the single-output line.

The I²C bus speed is specified at a maximum SCL rate of 100kHz SCL, which, admittedly, is not blazingly fast. The speed limit stems from the meager ability of a pullup resistor to source current to a long distributed line of peripherals. The 10-microsecond period allows plenty of time to charge the parasitic capacitance of the wires. (The maximum specified wire capacitance is 400 pF.)

PUTTING IT TOGETHER

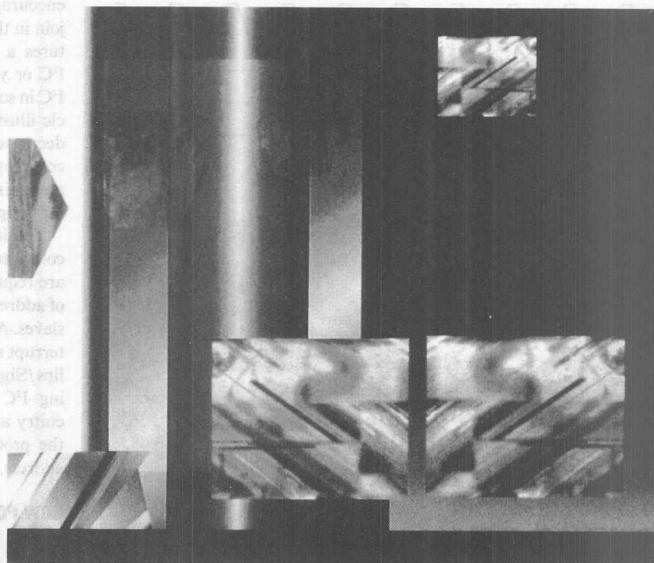
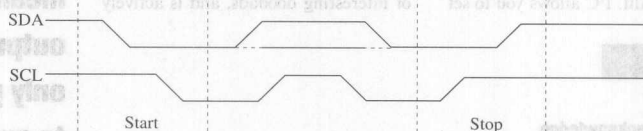
Although I²C supports multiple-master operation, here we use single-master, single-slave transactions to keep the example code simple. The master, as you might imagine, is defined as the unit that initiates the data transfer and generates the SCL signal. (In a multimaster system, each master would be responsible for generating its own SCL signal.) In our example, based strongly on the design of one of our company's single-board computers, the processor doesn't directly support I²C. Instead, we've implement-

ed the I²C bus using a couple of the pins on an 8255 peripheral I/O chip. Consequently, the bulk of the example application code is simple setup and house-keeping routines. (Steven R. Wheeler's example application listing was a bit too long to run in this issue. Interested readers may download it from the library 12 of CLMFORUM on CompuServe or from the Embedded Systems

Programming bulletin board service at (415) 905-2689—ed.)

By definition, a slave can be any processor or peripheral that responds to the master. Slaves all have unique, 7-bit addresses that are based on the device type and the wiring of address pins on the chip. All I²C peripherals have the top nibble of an address built in. For the PCF8574 I/O-port expanders we're us-

Figure 2
Start and stop conditions.



Exploring I²CExploring
I²C

ing as examples, the address is 0100xxx. The xxx indicates the address selected by the state of the three address pins on the peripheral.

I²C serial transactions are always eight bits of data from the transmitter followed by a ninth ACK bit from the receiver. The first step in any I²C data transfer is to send the address of the slave on the SDA line. This act might seem confusing, since we seem to be mixing 7-bit addresses with 8-bit data. In practice, it's quite easy to work with: addresses are always seven bits long, and the eighth bit is used to determine whether the operation is a read or a write. For example, upon transmitting 01000001 to the PCF8574, the slave, assuming it exists on the bus and is strapped to address 000, will respond with a low on the SDA line after the master has finished with its last (eighth) data bit. The master leaves the line high. If it doesn't find a slave with address 10000, the data line will remain high and a failed communication attempt can be detected.

If a slave is connected, it begins putting data on the SDA line as soon as it has detected that the eighth bit is set (which is a read request). The SDA line is driven to the data level when the SCL line is low. Data is read when SCL is high, so SDA must not change when SCL is high. This protocol leads to a

simple definition of the start of an I²C transaction—SDA goes from high to low when the clock is high.

The end of a transaction is equally simple to detect: SDA goes from low to high when SCL is high. This cycle leaves SDA and SCL in the high state, which is necessary if any other open-collector I²C peripheral wants access to the bus. Figure 2 illustrates the start and stop conditions of an I²C bus transaction.

ADDITIONAL DESIGN ROUTES

As you've seen, the I²C protocol is easy to work with and relatively simple to implement, even if you're not using a processor that directly implements it. If you're not planning to use Philips/Signetics microprocessors with onboard I²C support (such as the 68070 or various members of the 8051 family), you can still use the wide variety of available peripheral chips.

The number of integrated circuits using the I²C serial bus is increasing all the time. Application-oriented integrated circuits that support I²C include a voice synthesizer, a transcoder for IR remote control, several digital tuning circuits for computer-controlled television, several audio processors, PLL frequency synthesizers, tone generators, and frequency synthesizers. General-

purpose integrated circuits using I²C include LCD drivers, digital-to-analog converters, SRAMs, EEPROMs, and a RAM clock/calender.

I²C is very popular in Europe, where Philips has been aggressively marketing this flexible method of extending peripheral support to control projects, and it is currently catching fire on this side of the Atlantic. It seems reasonable to expect that, given the burden of printed-wire requirements for embedded systems based on increasingly wider chip buses, more and more designers seeking economy of means will be attracted to the economy of I²C.

Steven Sarns is the president of Vesta Technology in Wheat Ridge, Colo. He is a member of Mensa, Intel, and the Michigan Society of Professional Engineers. Sarns is also a founding member of the Denver chapter of the Forth Interest Group.

Jack Woehr is a senior project manager at Vesta Technology Inc. in Wheat Ridge, Colo. He is a Chapter Coordinator for the Forth Interest Group and is currently a member of the X3J14 Technical committee for ANS Forth. He can be reached by E-mail as jax@well.sf.ca.us or as VESTA on GENie.

Programming the I²C interface

Author: Mitchell Kahn

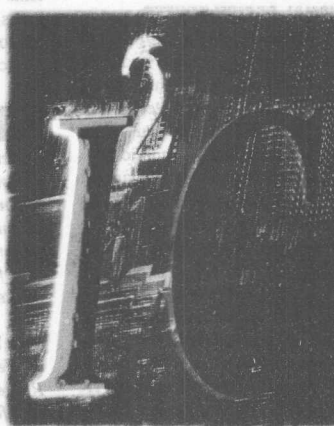
Dr. Dobb's JOURNAL

The Inter-Integrated Circuit Bus ("I²C Bus" for short) is a two-wire, synchronous, serial interface designed primarily for communication between intelligent IC devices. The I²C bus offers several advantages over "traditional" serial interfaces such as Microwire and RS-232. Among the advanced features of I²C are multimaster operation, automatic baud-rate adjustment, and "plug-and-play" network extensions.

Mention the I²C bus to a group of American engineers and you'll likely get hit with an abundance of blank stares. I say American engineers because until recently the I²C bus was primarily a European phenomenon. Within the last year, however, interest in I²C in the United States has risen dramatically. Embedded systems designers are realizing the cost, space, and power savings afforded by robust serial interchip protocols.

The idea of serial interconnect between integrated circuits is not new. Many semiconductor vendors offer devices designed to "talk" via serial links with other processors. Current examples include Microwire (National Semiconductor), SPI (Motorola), and most recently Echelon's Neuron chips. In all cases, the goal is the same: to reduce the wiring and pincount necessary for a parallel data bus. It simply does not make

Mitch is a senior strategic development engineer for Intel and can be contacted at 5000 W. Chandler Blvd., Chandler, AZ 85226 or at mkahn@sedona.intel.com.



economic sense to route a full-speed parallel bus to a slow peripheral.

Unfortunately for most serial-bus-capable devices, the choice of a bus protocol will dictate the CPU architecture. For example, only two CPU architectures implement an on-chip I²C port. If your choice of architecture precludes use of these architectures, then your only option is to implement the protocol in software.

The software implementation of the I²C protocol discussed in this article came about as a result of an implicit challenge during a staff meeting. One of our managers proposed that we hire a consultant to write a software I²C driver for the Intel 80C186EB embedded processor. Being somewhat new to the

group, I took exception (although not verbally!) to his suggestion. A weekend of intense hacking later, I presented the first prototype of the driver. My reward? I got to write a generic version of the driver for general distribution.

Design Trade-offs

Three distinct tasks are involved in implementing the I²C protocol: watching the bus, waiting for a specific amount of time, and driving the bus. This became apparent when I flowcharted 1 byte of a typical bus transaction; see Figure 1. The time delays associated with creating the bus waveforms would normally have been relegated to the 80C186EB's on-chip timers. I could not, however, assume that the end users of my code would be able to spare a timer for the software I²C port. I had to forego the elegance (and to some extent accuracy) of the on-chip timers for the sledgehammer approach of software timing loops. Luckily, the I²C protocol is extremely forgiving with regard to timing accuracy. The decision to use assembly instead of a high-level language stemmed directly from the need to control program-execution time. I had neither the time nor the inclination to hand-tune high-level code.

Having made the decision to use assembly language, I faced my next problem: Could I make the code portable? Intel offers a plethora of CPU and embedded-controller architectures. Would it be possible to make the code somewhat portable between disparate assembly languages? I found my answer in the use of macros.

Programming the I²C interface

All the basic building blocks of the I²C protocol (watching, waiting, and doing) can be compartmentalized into distinct macros. The algorithms that make up the I²C driver are written with these macros as the framework. You don't need to understand the intricacies of the I²C protocol to port these routines—you just need to know how to make your CPU watch, wait, and do.

For example, a 4.7- μ s delay is a common event during a transfer. The macro %Wait_4_7_uS implements just such a delay by using the 8086 LOOP instruction with a couple of NOPs for tuning; see Example 1(a). Total execution time is readily calculated from instruction timing tables. The same macro is ported to the i960 architecture in Example 1(b). Although I am a neophyte when it

comes to i960 programming, I had no problems porting the core macros.

Hardware Dependencies

A few words about the target hardware are in order before I discuss the code. Any implementation of the I²C protocol requires two open-drain (or open-collector), bidirectional port pins for the Serial Clock (SCL) and Serial Data (SDA) lines. The code in this article was designed for the 80C186EB embedded processor, which has two open-drain ports on-chip. The two pins, P2.6 (SCL) and P2.7 (SDA), are part of a larger 8-bit port. Processors without open-drain I/O ports can easily implement I²C with the addition of an external open-collector latch.

Two special-function registers, P2PIN and P2LTCH, are used to read and write the state of the port pins. The 80C186EB allows the special-function registers to be located anywhere in either memory or I/O space. For this implementation, I chose to leave the registers in I/O space, even though this limited my choice of instructions. The 80186 architecture does not provide for read-modify-write instructions in I/O space (an AND to I/O, for example); it can only load and store (IN and OUT). So why did I limit myself? Again, I had to assume the lowest common denominator for our customers when designing my code.

Building the Framework

Early on in development, I decided to partition my code macros according to physical processes involved in the I²C

protocol. Code not directly involved in mimicking the actions of a hardware I²C port was not written as macros. For example, the code necessary to access the stack frame is not written as a macro, whereas the code needed to toggle the clock line is. This was done to isolate architecture-dependent code sequences from the more generic I²C functions. Macros were also not used for "gray areas" such as the shifting of serial data, which is both architecture dependent and physical in nature. The I²C functions that passed the litmus test fell into the three aforementioned categories of watching, waiting, and doing.

The "waiting" macros provide a fixed-minimum time delay. They are implemented using a simple LOOP \$ delay. The LOOP instruction decrements the CX register, then branches to the target (in this case itself) if the result is non-zero. The delay is $(n-1) \times 15 + 5$ clocks, where n is the starting value in the CX register. All the delays were calculated assuming a 16-MHz clock rate (62.5 nanoseconds per clock). The code still works at lower CPU speeds because the I²C protocol only specifies minimum timings. In fact, the delay macros are only "accurate enough," providing timings as close as I could get to the specified minimum without undue tuning.

The "watching" macros are "spin-on-bit" polling loops. These pieces of code wait for a transition on the appropriate I²C line to occur before allowing execution to continue. There are two polling macros for each of the two I²C signal lines; one for high-to-low transitions and one for low-to-high transitions. The

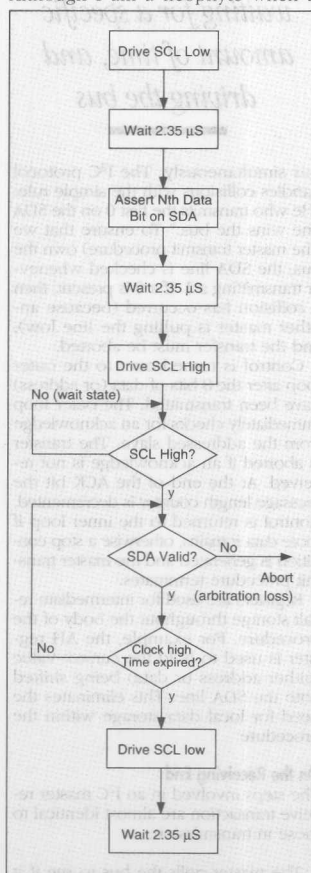
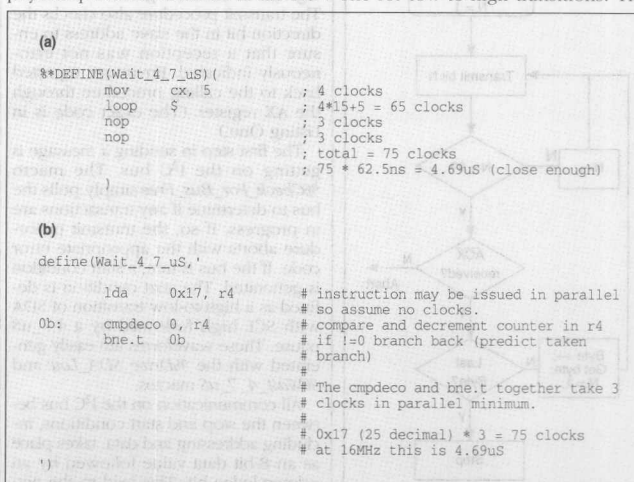


Figure 1: Flowchart of process for transmission of a single bit.



Example 1: (a) 80C186 implementation of 4.7- μ s wait macro; (b) 80960CA implementation of 4.7- μ s wait macro.

Programming the I²C interface

Programming the I²C interface

polling of the SCL line that gives rise to an important feature of I²C: automatic, bit-by-bit baud-rate adjustment. Any device on the I²C bus may hold the clock line low in order to stall the bus for more time (a serial wait state). The other devices on the bus are then forced to poll the SCL line until the slow device releases control of the clock.

The `%Get_SDA_Bit` macro also falls under the category of "watching." Its function is simply to return the state of the SDA line without waiting for a transition. `%Get_SDA_Bit` is used primarily to pull the serial data off the bus when the clock is valid.

The "doing" macros control the state of the clock and data lines. As with the polling macros, there are four types—one for each transition of the SCL or SDA lines. The "doing" macros are named to reflect the physical operations they perform. For example, `%Drive_SCL_Low` always drives the SCL line to a low state. `%Release_SCL_High`, on the other hand, relinquishes control of the SCL line, which may then be pulled high or driven low by another device on the bus. A read-modify-write operation is used for the bit manipulation so that the other 6 bits of Port 2 are not affected by the I²C operations.

Getting on the Bus

Three procedures were created using the macro framework. I'll describe only the master transmit (Listing One, page

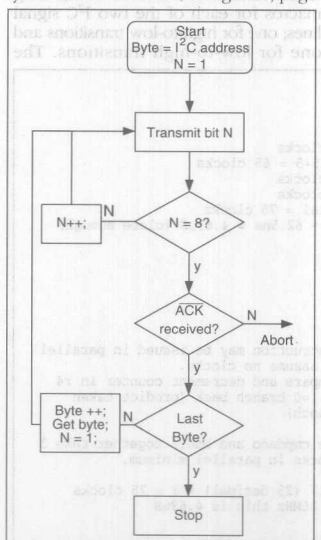


Figure 2: Flowchart for I²C transmit procedure.

106) and master receive functions (Listing Two, page 108), as they represent the needs of most I²C users. The slave procedure is long and intricate and will not be described here.

An I²C master transmission proceeds as follows:

1. The master polls the bus to see if it is in use.
2. The master generates a start condition on the bus.
3. The master broadcasts the slave address and expects an acknowledge (ACK) from the addressed slave.
4. The master transmits 0 or more bytes of data, expecting an ACK following each byte.
5. The master generates a stop condition and releases the bus.

The stack frame for the master transmit procedure, `I2CXA.A86`, includes a far pointer to the message for transmission, the byte count for the message, and the slave address. Far pointers and far procedure calls are used in all the procedures. No attempt was made to conform to a specific high-level language calling convention, although such a conversion would be trivial. The procedures save only the state of the modified segment registers.

The master transmit procedure performs error checking on the passed parameters before attempting to send the message. The maximum message length is set at 64 Kbytes by the segmentation of the 80186 memory space. This restriction could be removed by including code to handle segment boundaries. The transmit procedure also checks the direction bit in the slave address to ensure that a reception was not erroneously indicated. Errors are reported back to the calling procedure through the AX register. (The exact code is in Listing One.)

The first step in sending a message is getting on the I²C bus. The macro `%Check_For_Bus_Free` simply polls the bus to determine if any transactions are in progress. If so, the transmit procedure aborts with the appropriate error code. If the bus is free, a start condition is generated. The start condition is defined as a high-to-low transition of SDA with SCL high followed by a 4.7_μs pause. These waveforms are easily generated with the `%Drive_SDA_Low` and `%Wait_4_7_μs` macros.

All communication on the I²C bus between the stop and start conditions, including addressing and data, takes place as an 8-bit data value followed by an acknowledge bit. This led to the natural nested loop structure for the body of the procedure; see Figure 2.

The inner loop is responsible for transmitting the 8 bits of each data byte. Each transmitted bit generates the appropriate data (SDA) and clock (SCL) waveforms while checking for both serial wait states and potential bus collisions. A bus collision occurs when two masters attempt to gain control of the

*Three distinct tasks
are involved in
implementing the
I²C protocol:
watching the bus,
waiting for a specific
amount of time, and
driving the bus*

bus simultaneously. The I²C protocol handles collisions with the simple rule: "He who transmits the first 0 on the SDA line wins the bus." To ensure that we (the master transmit procedure) own the bus, the SDA line is checked whenever transmitting a 1. If a 0 is present, then a collision has occurred (because another master is pulling the line low), and the transfer must be aborted.

Control is turned over to the outer loop after the 8 bits of data (or address) have been transmitted. The outer loop immediately checks for an acknowledge from the addressed slave. The transfer is aborted if an acknowledge is not received. At the end of the ACK bit the message length counter is decremented. Control is returned to the inner loop if more data remains, otherwise a stop condition is generated and the master transmit procedure terminates.

Registers are used for intermediate result storage throughout the body of the procedure. For example, the AH register is used to hold the current value (either address or data) being shifted onto the SDA line. This eliminates the need for local data storage within the procedure.

On the Receiving End

The steps involved in an I²C master receive transaction are almost identical to those in transmission:

1. The master polls the bus to see if it is in use.
2. The master generates a start condi-

Programming the I²C interface

- tion on the bus.
3. The master broadcasts the slave address and expects an ACK from the addressed slave.
 4. The master receives 0 or more bytes of data and sends an ACK to the slave after each byte. The master signals the last byte by not sending an ACK.
 5. The master generates a stop condition and releases the bus.

A far pointer to the receive buffer is passed on the stack to the master receive procedure. The remainder of the parameters—slave address and message count—are identical between the two procedures. The received message length is fixed at 64 Kbytes, again because of segmentation. The error-checking, bus-availability sensing, and start-condition generation sections of the receive procedure are lifted verbatim from the transmit code.

The structure of the receive procedure differs slightly once the start con-

dition has been generated; see Figure 3. The slave address is transmitted using one iteration of the transmit procedure's outer loop. Control is passed to the receive loop once the slave acknowledges its address.

The receive loop structure is patterned after that of the transmit procedure. The inner loop controls the clocking of the SCL line and the shifting of the serial data off the SDA line into the CPU. Eight iterations of the inner loop are performed to receive each byte. The outer loop stores the received byte in the buffer, decrements the byte count, then sends an ACK to the slave. The last data byte is signalled by not sending an ACK.

Using the Procedures

Listing Three (page 110) shows a short program that uses both the master transmit and master receive procedures. The call to procedure I2C_XMIT displays the word "bUS-" on a four-character, seven-segment display controlled by the SAA1064 I²C compatible display driver. The time of day is read from the PCF8583 real-time clock by the call to procedure I2C_RECV.

Please note that interrupts must be disabled during the execution of both procedures. An interruption at an inopportune time (when the master is not in control of the clock) could cause the bus to hang. If you need to service interrupts periodically, then enable them only when the clock is driven low.

These procedures have been tested on a wide array of I²C devices ranging from serial EEPROMs to voice synthesizers. No compatibility problems have been seen to date.

Enhancing the Code

I've kicked around many ideas for enhancing the I²C procedures. You could,

for example, replace the timing loops with timed interrupts. That way, the CPU could perform useful work during the pauses. Along the same lines, the pauses could be scheduled using a real-time kernel, again improving CPU throughput. Finally, you could add a high-level language calling structure.

The use of timed interrupts adds an order of magnitude to the complexity of the code, but would be worth it for high-performance, real-time systems.

Conclusion

I²C is not the only game in town when it comes to serial protocols. Hopefully, some of the techniques presented here will carry over into the development of other "simulated" serial protocols, such as those targeted at the home-automation market. Who knows, maybe someday a snippet of my code may find its way into a truly intelligent dishwasher. I'll be waiting....

References

I²C Bus Specification, Philips Corporation (undated).

DDJ

Reprinted with permission of Dr.
Dobb's Journal, 1992

Entire contents copyright © 1992
by M&T Publishing, Inc.
Unless otherwise noted on specific articles.
All rights reserved.

ABP
American Business Press

The
Audit
Bureau

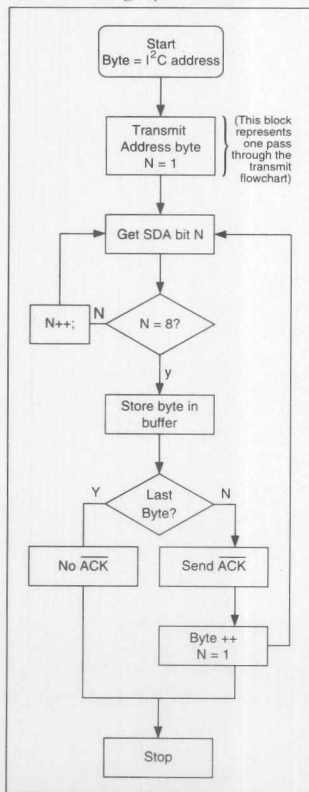


Figure 3: Flowchart for I²C receive procedure.

*All the basic
building blocks of
the I²C protocol
(watching, waiting,
and doing) can be
compartmentalized
into distinct macros*

Section 2

87C750, 8XC751, 8XC752 Application Notes

INDEX

AN422	Using the 8XC751 microcontroller as an I ² C bus master	See Section 1
AN423	Software driven serial communication routines for the 83C751 and 83C752 microcontrollers	2-3
AN426	Controlling air core meters with the 87C751 and SA5775	2-9
AN427	Timer 1 in non-I ² C applications of the 83/87C751/752 microcontrollers	2-23
AN428	Using the ADC and PWM of the 83C752/87C752	2-29
AN429	Airflow measurement using the 83/87C752 and "C"	2-36
AN430	Using the 8XC751/752 in multimaster I ² C applications	See Section 1
AN433	I ² C slave routines for the 83C751	See Section 1
AN436	"Opti-Mizer" power management for notebook computers using the 8XC752 microcontroller	2-56
AN439	87C751 fast NiCad charger	2-66
AN442	(BCM) 87C751 Specification for a bus-controlled monitor	2-78
AN445	ACCESS.bus mouse application code for the microcontroller	2-95
AN446	A software duplex UART for the 751/752	2-125
EIE/AN91007	I ² C driver routines for 8XC751/2 microcontrollers	See Section 1

Software driven serial communication routines for the 83C751 and 83C752 microcontrollers

AN423

DESCRIPTION

The need often arises to make use of a serial port in connection with a microcontroller that does not have a hardware UART on-chip. Aside from the obvious cases where the microcontroller application intrinsically requires RS-232 communications to achieve its purpose, a serial output may often be a simple and convenient method of providing detailed diagnostic information to the outside world while using only a single I/O port pin. In many cases, the solution may be to implement the UART function in software. The routines included here demonstrate a method to add such a function to a microcontroller without the benefit of a hardware UART.

Examples of microcontrollers that do not have on-chip UARTs are the 83C751 and 83C752. While it is possible to connect an external UART chip to these microcontrollers, it tends to use up many I/O port pins and begins to become less economical than simply using a standard 80C51. There are several factors to be considered in deciding if the software UART method will be usable in a particular application. The first is whether the serial communication channel is to be simplex (transmit only or receive only), half-duplex (transmit and receive, but not simultaneously), or full-duplex (simultaneous transmit and receive). Both simplex and half-duplex operation are fairly easy to implement in software on an 80C51-type microcontroller, and will be covered by this application note. Full-duplex operation is more difficult to implement in software and can use up a large portion of the microcontroller's time and resources.

A second consideration to be taken into account is the amount of system resources that will be "used up" by the serial communication software. First of all, such software routines will almost always require the use of at least one counter/timer to generate the time slices for the serial bit cells. Next, the physical connection to the outside world will require one I/O port pin each for the serial input and the serial output. Moreover, the port pin used for serial input should be an external interrupt input pin. This allows the software to be interrupted automatically at the beginning of an incoming start bit and synchronizes the timer accurately to the

serial data stream. Additional port pins may be used to implement signals such as Request to Send (RTS), Clear to Send (CTS), etc.

Finally, serial communication software will take up a certain amount of CPU time, more than would be required to operate a hardware UART. The overhead of software implemented serial communication may or may not be an issue, depending on the application, the throughput of the serial channel(s), the baud rate, other tasks the CPU is handling and how time-critical they are, etc.

The program listing that is included here is a demonstration of half-duplex serial routines on the 83C751 or 83C752 microcontrollers. The operation of the software would be the same on other 80C51 derivatives, except that the counter/timer operation is slightly different. The program, as listed, will send a canned message to the serial output (port pin P1.0 in this case), then wait for data on the serial input (port pin P1.5/INT0). When a character has been received on the serial input, it will be echoed through the serial output. Since the software is inherently half-duplex, the rate at which characters are received must be less than half the rate that would be possible on a full-duplex channel. This example has been set up to receive and transmit at 9600 baud when run with a 16 MHz crystal.

The operation of the routines is fairly straightforward. Beginning with a start bit occurring on the serial input line, an interrupt (external interrupt 0) will occur. At the interrupt service routine Int0, the counter/timer is loaded with a value that will result in a time delay that is approximately equivalent to half a bit cell time for the baud rate being used, less some constant to account for the elapsed time between a timer interrupt and the point where the serial input is actually sampled. The timer reload register is loaded with a value that will result in a time delay that is as close as can be calculated to one full bit cell time. The program then starts the timer and simply returns to the main program, waiting for the timer to time out, generating another interrupt.

At that point, the serial start bit should be about halfway through its nominal duration.

When the first timer interrupt occurs, the timer interrupt routine Timr0 calls the receive bit routine RxBit which checks to make sure that the start bit is still valid and flags an error if it is not. The RxBit routine will then return control to the main program routine, waiting for the next timer interrupt.

On the second timer interrupt, the RxBit routine reads the serial input line and shifts the value into the serial holding register RxDat. This process is repeated until 8 bits have been read in on consecutive timer interrupts. Finally, on the tenth timer interrupt, the receive routine looks for a valid stop bit and flags an error if one is not detected. At this point, the RcvRdy flag is set to inform the main program that a character is waiting in the holding register.

The transmit routine works in a somewhat similar fashion, beginning with a call to the byte transmit routine XmtByte, which first checks to make sure that a byte receive operation is not already in progress. The RSXmt routine will then set up the timer and timer reload registers to correspond to one bit cell time, start the timer, and assert a start bit.

At each subsequent timer interrupt, the routine TxBit shifts out the next bit from the transmit holding register XmtDat, until all 8 bits have been transmitted. Once all of the data has been sent, the stop bit is asserted on the next timer interrupt. A final timer interrupt is required to insure that the stop bit lasts at least one full bit cell time. At this point, transmit flag TxFlag is cleared in order to inform the main program that the transmission is completed.

A few other useful routines are embedded in the sample program: PrByte, which converts a byte of data to hexadecimal form and transmits it; HexAsc, which converts one nibble of raw data to hexadecimal form; and Mess, which transmits an absolute string of data (usually a text message) which is terminated by a 0 byte.

This demonstration of software driven serial port routines uses 5 bytes of microcontroller RAM, two port bits (including one external interrupt input), one counter/timer, and about 256 bytes of code space, excluding the message string at the end of the listing.

Software driven serial communication routines for the 83C751 and 83C752 microcontrollers AN423

```

RS751 Half-Duplex Serial Communication Routines
1
2 *****
3
4 Software Driven Half-Duplex Serial Communication Routines
5 for 83C751 and 83C752 series Microcontrollers
6
7
8
9 *****
10
11 $Title(Half-Duplex Serial Communication Routines)
12 $Date(11/14/89)
13 $MOD751
14
15 *****
16
17 FF75 BaudVal EQU -139 ;Timer value for 9600 baud @ 16 MHz.
18 ;(one bit cell time)
19 FFD9 StrtVal EQU -39 ;Timer value to start receive.
20 ;(half of one bit cell time, minus the
21 ;time it takes the code to sample RxD)
22
23 0010 XmtDat DATA 10h ;Data for RS-232 transmit routine.
24 0011 RcvDat DATA 11h ;Data for RS-232 receive routine.
25 0012 BitCnt DATA 12h ;RS-232 transmit & receive bit count.
26 0013 LoopCnt DATA 13h ;Loop counter for test routine.
27
28 0020 Flags DATA 20h
29 0000 TxFlag BIT Flags.0 ;Receive-in-progress flag.
30 0001 RxFlag BIT Flags.1 ;Transmit-in-progress flag.
31 0002 RxErr BIT Flags.2 ;Receiver framing error.
32 0003 RcvRdy BIT Flags.3 ;Receiver ready flag.
33
34 0090 TxD BIT P1.0 ;Port bit for RS-232 transmit.
35 0095 RxD BIT P1.5 ;Port bit for RS-232 receive (INT0).
36
37 *****
38
39 ; Interrupt Vectors
40
41 0000 ORG 0 ;Reset vector.
42 0000 0124 AJMP Reset
43
44 0003 ORG 03h ;External interrupt 0.
45 0003 019F AJMP ExInt0 ;Indicates RS-232 start bit received.
46
47 000B ORG 0Bh ;Timer 0 interrupt.
48 000B 0175 AJMP Timr0 ;Baud rate generator.
49
50 0013 ORG 13h ;External interrupt 1. (not used).
51 0013 32 RETI
52
53 001B ORG 1Bh ;Timer I interrupt (not used).
54 001B 32 RETI
55
56 0023 ORG 23h ;I2C interrupt (not used).
57 0023 32 RETI
58

```

Software driven serial communication routines for the 83C751 and 83C752 microcontrollers

AN423

```

59 ;*****
60
61 ;Simple test of RS-232 transmit and receive.
62
0024 758130 63 Reset: MOV SP,#30h
0027 752000 64 MOV Flags,#0 ;Clear RS-232 flags.
002A C201 65 CLR RxFlag ;
002C 758800 66 MOV TCON,#00h ;Set up timer controls.
002F 75A882 67 MOV IE,#82h ;Enable timer 0 interrupts.
68 ;End of setup routine.
0032 751310 69 MOV LoopCnt,#16 ;Test transmit first.
0035 7900 70 MOV R1,#0 ;Zero line count.
0037 90010C 71 MOV DPTR,#Msg1 ;Point to message string.
003A 11FB 72 Loop1: ACALL Mess ;Send an RS-232 message repeatedly.
003C 743A 73 MOV A,#':' ;
003E 1154 74 ACALL XmtByte ;
0040 E9 75 MOV A,R1 ;
0041 11DD 76 ACALL PrByte ;Print R1 contents.
0043 09 77 INC R1 ;Advance R1 value.
0044 D513F3 78 DJNZ LoopCnt,Loop1 ;
79 ;End of test routine.
0047 D2A8 80 Loop2: SETB EX0 ;Enable interrupt 0 (RS-232 receive).
0049 3003FD 81 JNB RcvRdy,$ ;Wait for data available.
004C C203 82 CLR RcvRdy ;
004E E511 83 MOV A,RcvDat ;Echo same byte.
0050 1154 84 ACALL XmtByte ;
0052 80F3 85 SJMP Loop2 ;
86 ;First timeout wait a bit time.
87 ;
88 ; Send a byte out RS-232 and wait for completion before returning.
89 ; (use if there is nothing else to do while RS-232 is busy)
90 ;
91 ;
0054 2001FD 91 XmtByte: JB RxFlag,$ ;Wait for receive complete.
0057 115D 92 RSXmt: ACALL RSXmt ;Send ACC to RS-232 output.
0059 2000FD 93 JB TxFlag,$ ;Wait for transmit complete.
005C 22 94 RET ;
95 ;
96 ;
97 ; Begin RS-232 transmit.
98 ;
005D F510 99 RSXmt: MOV XmtDat,A ;Save data to be transmitted.
005F 75120A 100 MOV BitCnt,#10 ;Set bit count.
0062 758CFF 101 MOV TH,#High BaudVal ;Set timer for baud rate.
0065 758A75 102 MOV TL,#Low BaudVal ;
0068 758DFF 103 MOV RTH,#High BaudVal ;Also set timer reload value.
006B 758B75 104 MOV RTL,#Low BaudVal ;
006E D28C 105 SETB TR ;Start timer.
0070 C290 106 CLR TxD ;Begin start bit.
0072 D200 107 SETB TxFlag ;Set transmit-in-progress flag.
0074 22 108 RET ;
109 ;
110 ;
111 ; Timer 0 timeout: RS-232 receive bit or transmit bit
112 ;
0075 C0E0 113 Timr0: PUSH ACC ;
0077 C0D0 114 PUSH PSW ;
0079 20013E 115 RxBit: JB RxFlag,RxBit ;Is this a receive timer interrupt?
007C 200007 116 JB TxFlag,TxBit ;Is this a transmit timer interrupt?

```

```

007F C28C 117 T0Ex1: CLR TR ;Stop timer.
0081 D0D0 118 T0Ex2: POP PSW
0083 D0E0 119 POP ACC
0085 32 120 RETI
121
122 ; RS-232 transmit bit routine.
123
124
0086 D51204 125 TxBit: DJNZ BitCnt,TxBusy ;Decrement bit count, test for done.
0089 C200 126 CLR TxFlag ;End of stop bit, release timer.
008B 80F2 127 SJMP T0Ex1 ;Stop timer and exit.
128
008D E512 129 TxBusy: MOV A,BitCnt ;Get bit count.
008F B40104 130 CJNE A,#1,TxNext ;Is this a stop bit?
0092 D290 131 SETB TxD ;Set stop bit.
0094 80EB 132 SJMP T0Ex2 ;Exit.
133
0096 E510 134 TxNext: MOV A,XmtDat ;Get data.
0098 13 135 RRC A ;Advance to next bit.
0099 F510 136 MOV XmtDat,A
009B 9290 137 MOV TxD,C ;Send data bit.
009D 80E2 138 SJMP T0Ex2 ;Exit.
139
140
141 ;Begin RS-232 receive (after external interrupt 0).
142
009F 75120A 143 ExInt0: MOV BitCnt,#10 ;Set receive bit count.
00A2 758CFF 144 MOV TH,#High StrtVal ;First timeout in HALF a bit time.
00A5 758AD9 145 MOV TL,#Low StrtVal
00A8 758DFF 146 MOV RTH,#High BaudVal ;Set timer reload for baud rate.
00AB 758B75 147 MOV RTL,#Low BaudVal
00AE 751100 148 MOV RcvDat,#0 ;Initialize received data to 0.
00B1 C2A8 149 CLR EX0 ;Disable external interrupt 0.
00B3 C202 150 CLR RxErr ;Clear error flag.
00B5 D28C 151 SETB TR ;Start timer.
00B7 D201 152 SETB RxFlag ;Set receive-in-progress flag.
00B9 32 153 RETI
154
155
156 ; RS-232 receive bit routine.
157
00BA D5120D 158 RxBit: DJNZ BitCnt,RxBusy ;Decrement bit count, test for stop.
00BD 209502 159 JB RxD,RxBtEx ;Valid stop bit?
00C0 D202 160 RxBtErr: SETB RxErr ;Bad stop bit, tell mainline.
00C2 C201 161 RxBtEx: CLR RxFlag ;Release timer for other purposes.
00C4 D2A8 162 SETB EX0 ;Re-enable external interrupt 0.
00C6 D203 163 SETB RcvRdy ;Tell mainline that a byte is ready.
00C8 80B5 164 SJMP T0Ex1 ;Stop timer and exit.
165
00CA E512 166 RxBusy: MOV A,BitCnt ;Get bit count.
00CC B40905 167 CJNE A,#9,RxNext ;Is this a start bit?
00CF 2095EE 168 JB RxD,RxBtErr ;Valid start bit?
00D2 80AD 169 SJMP T0Ex2 ;Exit.
170
00D4 E511 171 RxNext: MOV A,RcvDat ;Get partial receive byte.
00D6 A295 172 MOV C,RxD ;Get receive pin value.
00D8 13 173 RRC A ;Shift in new bit.
00D9 F511 174 MOV RcvDat,A ;Save updated receive byte.

```


Software driven serial communication routines for the 83C751 and 83C752 microcontrollers

AN423

```

00DB 80A4      175      SJMP      T0Ex2      ;Exit.
176
177
178      ; Print byte routine: print ACC contents as ASCII hexadecimal.
179
00DD C0E0      180      PrByte:  PUSH  ACC
00DF C4        181      SWAP   A
00E0 11EB      182      ACALL  HexAsc
00E2 1154      183      ACALL  XmtByte
00E4 D0E0      184      POP    ACC
00E6 11EB      185      ACALL  HexAsc      ;Print nibble in ACC as ASCII hex.
00E8 1154      186      ACALL  XmtByte
00EA 22        187      RET
188
189
190      ; Hexadecimal to ASCII conversion routine.
191
00EB 540F      192      HexAsc:  ANL    A,#0FH      ;Convert a nibble to ASCII hex.
00ED 30E308    193      JNB     ACC.3,NoAdj
00F0 20E203    194      JB      ACC.2,Adj
00F3 30E102    195      JNB     ACC.1,NoAdj
00F6 2407      196      Adj:    ADD    A,#07H
00F8 2430      197      NoAdj:  ADD    A,#30H
00FA 22        198      RET
199
200
201      ; Message string transmit routine.
202
00FB C0E0      203      Mess:    PUSH  ACC
00FD 7800      204      MOV     R0,#0      ;R0 is character pointer (string
00FF E8        205      Mes1:    MOV     A,R0      ;length is limited to 256 bytes).
0100 93        206      MOVVC   A,@A+DPTR      ;Get byte to send.
0101 B40003    207      CJNE    A,#0,Send      ;End of string is indicated by a 0.
0104 D0E0      208      POP     ACC
0106 22        209      RET
210
0107 1154      211      Send:    ACALL  XmtByte      ;Send a character.
0109 08        212      INC     R0      ;Next character.
010A 80F3      213      SJMP     Mes1
214
010C 0D0A      215      Msg1:    DB      0Dh, 0Ah
010E 54686973  216      DB      'This is a test of the software serial routines.', 0
0112 20697320
0116 61207465
011A 7374206F
011E 66207468
0122 6520736F
0126 66747761
012A 72652073
012E 65726961
0132 6C20726F
0136 7574696E
013A 65732E00
217
218      END

```

ASSEMBLY COMPLETE, 0 ERRORS FOUND

Software driven serial communication routines for the 83C751 and 83C752 microcontrollers

AN423

ACC.	D	ADDR	00E0H	PREDEFINED		175	0000 8004
ADJ.	C	ADDR	00F6H			176	
BAUDVAL.		NUMB	FF75H			177	
BITCNT.	D	ADDR	0012H			178	
EX0.	B	ADDR	00A8H	PREDEFINED		179	
EXINT0.	C	ADDR	009FH			180	
FLAGS.	D	ADDR	0020H			181	
HEXASC.	C	ADDR	00EBH			182	
IE.	D	ADDR	00A8H	PREDEFINED		183	
LOOP1.	C	ADDR	003AH			184	
LOOP2.	C	ADDR	0047H			185	
LOOPCNT.	D	ADDR	0013H			186	
MESL.	C	ADDR	00FFH			187	
MESS.	C	ADDR	00FBH			188	
MSG1.	C	ADDR	010CH			189	
NOADJ.	C	ADDR	00F8H			190	
P1.	D	ADDR	0090H	PREDEFINED		191	
PRBYTE.	C	ADDR	00DDH			192	
PSW.	D	ADDR	00D0H	PREDEFINED		193	
RCVDAT.	D	ADDR	0011H			194	
RCVRDY.	B	ADDR	0003H			195	
RESET.	C	ADDR	0024H			196	
RSXMT.	C	ADDR	005DH			197	
RTH.	D	ADDR	008DH	PREDEFINED		198	
RTL.	D	ADDR	008BH	PREDEFINED		199	
RXBIT.	C	ADDR	00BAH			200	
RXBITE.	C	ADDR	00C2H			201	
RXBTE.	C	ADDR	00C0H			202	
RXBUSY.	C	ADDR	00CAH			203	
RXD.	B	ADDR	0095H			204	
RXERR.	B	ADDR	0002H			205	
RXFLAG.	B	ADDR	0001H			206	
RXNEXT.	C	ADDR	00D4H			207	
SEND.	C	ADDR	0107H			208	
SP.	D	ADDR	0081H	PREDEFINED		209	
STRTVAL.		NUMB	FFD9H			210	
TOEX1.	C	ADDR	007FH			211	
TOEX2.	C	ADDR	0081H			212	
TCON.	D	ADDR	0088H	PREDEFINED		213	
TH.	D	ADDR	008CH	PREDEFINED		214	
TIMR0.	C	ADDR	0075H			215	
TL.	D	ADDR	008AH	PREDEFINED		216	
TR.	B	ADDR	008CH	PREDEFINED		217	
TXBIT.	C	ADDR	0086H			218	
TxBusy.	C	ADDR	008DH			219	
TXD.	B	ADDR	0090H			220	
TXFLAG.	B	ADDR	0000H			221	
TXNEXT.	C	ADDR	0096H			222	
XMTBYTE.	C	ADDR	0054H			223	
XMTDAT.	D	ADDR	0010H			224	

Controlling air core meters with the 87C751 and SA5775 AN426

Author: Bill Houghton

INTRODUCTION

Often, certain classes of microcontroller applications surface where large amounts of on-chip resources such as a large program memory space and numerous I/O pins are not required. These applications are typically cost sensitive and desirable attributes of the MCU include low cost and modest on-chip resources such as program and data memory, I/O, and timer-counters. Substantial benefits of reduced design cycle time can be realized by using an industry-standard architecture having software compatibility with existing popular microcontrollers.

THE 87C751

The Philips 87C751 is one such microcontroller that easily meets these requirements. This device, shown in Figure 1, has a 2k x 8 program memory, 64 bytes of RAM, 19 parallel I/O lines, and a 16-bit autoreload timer-counter. It also includes an I²C serial interface and a fixed rate timer. The 87C751 is based on the 80C51 core and thus uses an industry-standard architecture and instruction set. The device is available in both ROM (83C751) and EPROM (87C751) versions. The EPROM version is available in both UV erasable and OTP packages. References to the 87C751 in this document also apply to the 83C751, unless explicitly stated.

TYPICAL APPLICATION

A typical example of such an application is the interface between the 87C751 and the Philips SA5775 Air Core Meter Driver shown in Figure 2. This circuit includes the 87C751 microcontroller, the SA5775 air core meter driver, an NE555 timer, and discrete support components.

An air core meter differs from a conventional (d'Arsonval) meter movement in that it has no spring to return the needle to a predetermined position, no zeroing adjustment, and no permanent magnet in the classical sense. Instead, it consists of two coils of wire wound in quadrature with each other around a central core in which there is a disc magnetized along its diameter. A shaft is placed through the center of this disc so that the shaft rotates with the disc. An indicating needle attached to this shaft will rotate with it.

SA5775 Air Core Meter Driver

The SA5775 is a monolithic driver for controlling air core meters typically used in automotive instrument clusters and is shown in Figure 3. The SA5775 receives a 10-bit

serial word and converts that word to four voltage outputs that appear at the SINE+, SINE-, COSINE+, and COSINE- outputs. The differential voltage at the SINE outputs are applied to one coil of the meter and the COSINE outputs are applied to the other coil of the meter.

The currents through these coils produce a resultant magnetic force which is the vector sum of the magnetic forces produced by each of the two coils. Since the currents through the coils are bidirectional this magnetic vector can rotate through a full 360 degrees. The magnetized disc within the air core meter will follow the rotating vector and the needle will indicate the vector's current position. Since 10 bits are used, there are 1024 discrete words available resulting in an angular displacement of 0.3516 degrees per bit. This is small enough to provide an apparently smooth movement of the needle. The smoothness of the motion will depend greatly on the damping factor of the meter movement.

A simplified block diagram of the SA5775 is shown in Figure 4. This device consists of a serial-in/parallel-out shift register, a data latch, a D/A converter, buffers, and an internal voltage reference.

A logic high must be present on the chip select (CS) input to clock in the data. Data appearing on the data input (DI) pin is clocked into the shift register on the rising edge of the clock (CLK) input. The data output (DO) pin is the overflow from the shift register, allowing the user to daisy chain multiple SA5775 devices. Note that data is clocked out of this pin on the falling edge of the clock. The CS pin is also used to latch the parallel outputs of the shift register into the data latch. The outputs of the data latch feed the inputs to the D/A converter. The D/A converter outputs are buffered to form the drive signals for the meter coils.

A voltage reference for the D/A converter is provided internally. It is possible to externally force different values for these voltages and pins are provided for this purpose. However, this is not generally recommended as this could lead to increased power dissipation.

The D/A converter circuits and its associated output buffers are purposely designed such that the span of these circuits does not include the power supply rails. This is to avoid inaccuracies that would otherwise occur if the output were to become very close to either supply rail. With a supply voltage of 14 volts (VIGN), the positive reference

(VREF+) is approximately 8 volts and the negative reference (VREF-) is approximately 1 volt. The outputs will then span a range from 1 volt to approximately 11 volts. The maximum output is $[(VREF+) + 0.41((VREF+) - (VREF-))]$. The SA5775 is designed to drive air core meters having a minimum winding impedance of 200 ohms.

The clock high and low time requirements are each 200 ns minimum, implying a maximum data rate approaching 2.5 megabits per second. At this rate it would require approximately 4 ms to ramp from zero to full scale if all binary codes were loaded into the SA5775. However, the air core meter cannot respond to such data rates.

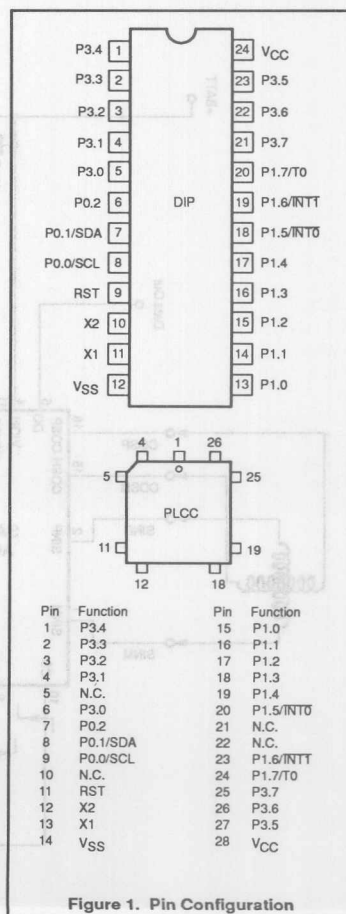
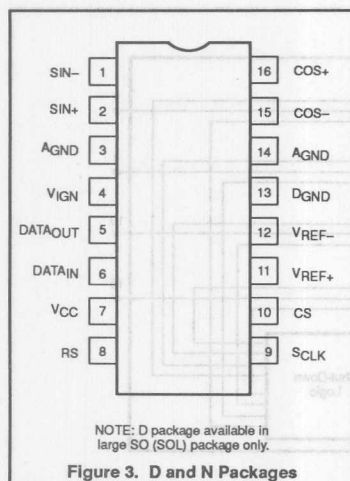


Figure 1. Pin Configuration

Controlling air core meters with the 87C751 and SA5775 AN426

**87C751 Microcontroller**

The 87C751 microcontroller provides all of the intelligence in this application. It samples various input ports to determine which demonstration programs to run, the incremental step sizes for angular displacement of the meter core, and the time delay between increments. In one of the demonstration modes, it also samples a variable frequency input and positions the meter core in response to the frequency of that input. The 87C751 also transmits the 10-bit serial data to the SA5775. Data input (DI), Clock (CLK), and Chip Select (CS) lines are driven from the 87C751.

Port 0 of the 87C751 is a 3-bit wide port and is used for communicating data to the ACMD. Data is transmitted, MSB first, in a serial stream clocked into the DI of the SA5775 on the rising edge of the clock. In order to clock in data, the CS pin of the SA5775 must be high. The data in the input register is shifted into a latch that drives the DAC on the high to low transition of the CS line. As data is shifted into the ACMD, it overflows through the Data Out (DO) pin on the falling edge of the clock. With this facility, multiple ACMDs can be daisy-drained with DO of one ACMD being connected to DI of the next one, and common clock and chip select lines may be used. This simplifies the interfacing to multiple meter drivers.

The 78L05 regulator (Q2) provides 5 Volt power for the board so that single supply of +14 volts can be applied to the board.

Three rotary switches are used on this board. The PROGRAM SELECT switch (S3) is used to select the program routine that is

executed, the INC SELECT (S2) switch selects the incremental step sizes of the two of the routines, and the DELAY switch (S4) is used to set the delay between successive word transmissions in one of the routines.

The START/COUNT button (S5) is used to begin execution of a routine, and to cause the next incremental step in Routine #1.

The COUNT UP/DOWN switch (S6) is used in Routine #1 to determine whether the count is increased or decreased with transmission of successive words.

NE555 Timer

The NE555 timer shown in this application example is used as a free running squarewave generator used to simulate sensor inputs such as those which might be found in an automobile, etc. The NE555 timer (U4) operates in the astable mode to produce an output frequency that can be varied from about 1Hz to about 200 Hz. Three of the program routines measure the input period and produce an output code that is proportional to the frequency present at pin 20 (TO) of the microcontroller. A RATE switch (S7) is used to select between the on board oscillator or an external source.

The program listing is included at the end of this application note.

Program Entry

The program starts at address 030(hex) on line 21 of the program listing. The first task is to write 1's to all pins of each port.

Lines 25 and 26 clear registers 6 and 7.

These registers are used in this program only to hold the data that is sent out to the ACMD. The registers are cleared to be sure that the starting value is zero.

At line 27 the program waits until the START/COUNT button (S5) is depressed before continuing. Lines 28 and 29 set the timer to overflow after 10ms. This is done by setting the timer registers for a count of 10,000 microseconds less than full scale. When the timer counter overflows the timer flag is set, and the timer is reloaded with the value in the timer register. By examining the timer flag we know when 10ms has expired.

Line 30 calls subroutine RPS (Read Port Selected), which reads Port 3 to determine which routine has been selected. Since the PROGRAM SELECT switch (S3) is connected to port pins P3.2 through P3.4, subroutine RPS (lines 507 through 511 at the end of the program) first reads Port 3 into the accumulator, then complements it because the switches used are complementary binary. The reading is then rotated right once and the upper nibble and the LSB (least significant

bit) are masked off, leaving twice the value of the port selected in the accumulator. Twice the read value is needed for the next few main program lines that determine which routine to execute.

Line 31 moves the address of label JMPTBL (Jump Table) to the 16-bit Data Pointer (DPTR) register. Line 32 causes a program jump to the address that is the sum of the value in the accumulator (two times the routine number selected) plus the DPTR register. Since each of the commands on lines 33 through 40 are two byte commands, these addresses are all separated by two bytes; hence, the need for the accumulator to contain a number that is twice the number of the selected routine.

Routine 0

This routine begins on line 41 by incrementing the 10-bit word in registers 7 and 6 by the amount indicated by the setting of the INCREMENT SELECT switch, then sending that word to the SA5775. When a full scale overflow is detected, a full scale code (3FF hex) is sent out, followed by a delay of 500 ms, then successive output codes are sent out, decremented by an amount indicated by the INCREMENT SELECT switch. When an underflow is detected a code of zero scale is sent and the routine returns to the beginning of the program. This routine is implemented with a series of subroutine calls.

The SO subroutine begins on line 356 and starts by sending out whatever ten bits that in the two LSBs of register 7 (R7) plus the 8 bits of R6 by calling the SENDIT subroutine. Then it calls the UP subroutine, which increases the word value to be sent out. The program then jumps to the beginning of this subroutine, repeating the process of sending out a word and incrementing to the next word until an overflow from the tenth bit (bit 2 of R7) is detected at line 362.

The SENDIT subroutine (beginning on line 476) brings the CS line high, sets a bit counter (R1) to 2 (to send out two bits of R7), brings the value of R7 to the accumulator, rotates the accumulator to the right three times through the carry bit to bring the two LSBs to the position of the two MSBs, calls the SEND1 routine, which sends the number of bits in the accumulator, starting with the MSB, indicated by R1. Counter R1 is then set to 8 to send out all 8 bit of R6 and the accumulator is loaded with the contents of R6. The SEND1 routine is again called to send out the final 8 bits, and, on line 491, the CS line is brought low, loading the SA5775 internal parallel latch with the contents of the input shift register.

Controlling air core meters with the 87C751 and SA5775

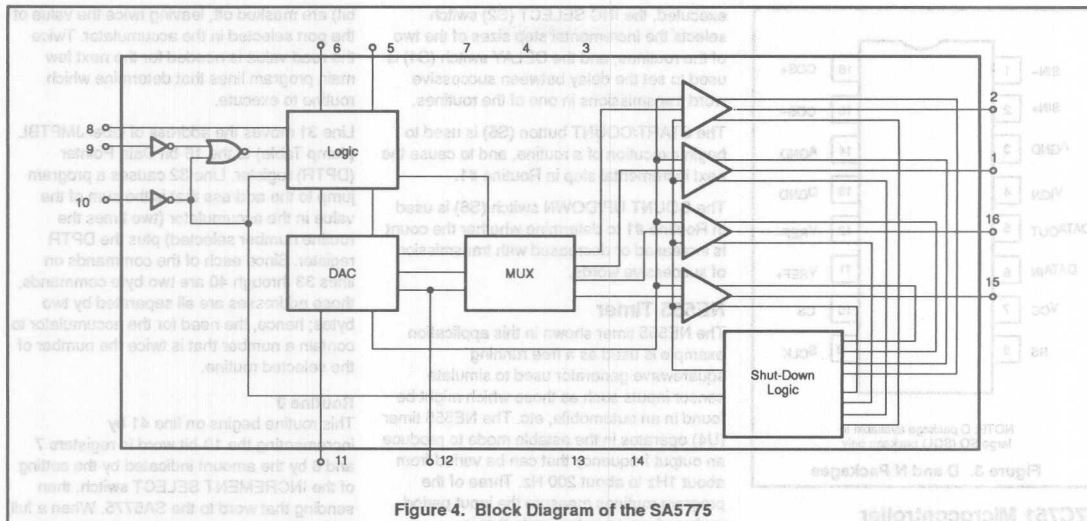


Figure 4. Block Diagram of the SA5775

The SEND1 routine rotates the accumulator left through the carry bit, moves the value of the carry bit to port pin PO.1 (SDA—Serial Data pin), waits to provide a setup time, brings the clock low, waits, brings the clock high, waits, then decrements bit counter sends the next bit if the counter is not zero. A return is executed when the counter becomes zero.

The UP subroutine, beginning at line 364, reads the delay selected by switch S4 at port pin P1, complements it (again, because the rotary switches are complementary binary), masks off the upper four bits (because the delay switch has just four positions and is connected to the lower four bits of the port), multiplies it by 4 (rotates left twice), then moves the result to R1. If R1 is not zero, the program jumps around line 376 and calls a 10ms delay (subroutine DLY10MS) the number of times entered into R1.

The 10ms delay subroutine (starting at line 436) sets the timer for 10ms, waits at line 446 for the timer flag to be set, clears the timer flag, stops the timer, and returns, in this case, to line 379, where the program decrements R1 and repeats the 10ms delay until R1 is zero.

If the selected delay was zero, the program jumps from line 376 to line 380 and reads port 3 to determine the amount the sent out word is to change from the value previously sent out. The accumulator is complemented and the upper 6 bits masked off to recover only the two bits of the selected increment amount. Since increments of 1, 2, 3, or 4

LSBs are hardly noticeable, the program then multiplies the result by 8 (rotate left three times). To insure a minimum change amount, the accumulator is incremented by one at line 386. This all means that the increment amounts that can be selected are 1, 9, 17, or 25 LSBs. This amount is added, in lines 387 through 391, to the word previously sent out and we return from this subroutine.

After calling the S0 subroutine, PR0GO call the FULLSC (full scale) subroutine, which sends out the full scale code of 3E8(hex). Although a 10-bit full scale code would be 3FF(hex), going only to 3E8 allows an easy distinction between zero scale and full scale when looking at the display. The FULLSC subroutine is found at line 352.

After advancing to full scale, there is a 500ms delay, found at line 464 and called from line 48, then 49 calls the S0D subroutine to send out decreasing word values.

The S0D subroutine begins at line 393 and begins by sending out the current word in R7 and R6 from line 398, then calling subroutine DOWN, which calculates the next (decreasing) word to send out. DOWN begins at line 402. It essentially does the same thing as the UP subroutine, but subtracts the INCREMENT SELECT value from the previously sent word rather than adding to it.

At line 50 subroutine ZEROSC is called to send a zero scale code to the SA5775, then the program branches back to the beginning.

Routine 1

This routine is selected with the PROGRAM SELECT switch is in position 1 or position 9. Routine 1 (PROG1) increments or decrements the word sent out, depending upon the setting of the COUNT UP/COUNT DOWN switch, S6. The amount of change is determined by the setting of the INC SELECT switch, S2.

At line 63, the program examines S6 at port pin P3.6 and jumps to the decrement portion of the routine if the pin is low. If this pin is high, the UP subroutine is called from line 64 to increase the R7/R6 word value. The UP subroutine was previously described.

If pin P3.6 is low, the DOWN subroutine (line 402) decreases the previous word sent out by the amount determined from the INC SELECT switch setting.

To insure enough delay to allow the user time to release the START/COUNT button (S5), a delay of 200ms is included at line 66 before jumping to line 27, where another depression of the START/COUNT button is awaited. If S3 (PROGRAM SELECT) is still set to 1 or 9, depression of S5 will cause a jump back to line 52. If another program is selected, the program will jump to the selected routine.

Holding down S5 with PROGRAM SELECT set at position 1 or 9 will cause increasing or decreasing word values to be sent to the SA5775.

Controlling air core meters with the 87C751 and SA5775 AN426

Routine 2

PROG2 is the most complex of all these routines. The purpose of this routine is to cause the air core meter deflection to represent the frequency presented at the timer/counter input to the microcontroller. This is done by measuring the period of the input square wave and taking the inverse of the period. The input here must be a square wave because a slow rise and fall time at this input will cause fluctuating readings. To determine the frequency by counting pulses for a time would require a much longer time and, therefore, is impractical.

The MEAS (measure) subroutine is called at line 79 to measure the period of the input waveform and the CALC (calculate) subroutine is called at line 80 to calculate the code to send to the SA5775. The SENDIT subroutine is then called to send the word to the SA5775 and the program jumps back to line 28.

The MEAS subroutine begins at line 83 by being sure the timer is not running and clearing the timer (overflow) flag, then entering zero into both high and low bytes of the timer and the timer register. The carry bit is then cleared (line 90) and the timer started and the timer interrupt enabled.

Lines 93 and 94 form a short loop that waits until either the carry bit is set or until the TO input is low. The carry bit is set when the timer has gone beyond one second. This is done by the timer interrupt subroutine, found at lines 16 through 19. If the TO input never goes low, we know the frequency is at or near zero and the program jumps to GZS (line 108) where R3 is loaded with a 1F (hex) to cause the CALC subroutine to load zero scale into R7/R6.

When (and if) TO is found to be low, the program jumps to line 95 and waits for that input to go high. Time out process is the same as above.

Now that the TO input is found high (if is before the one second time out), the timer and carry bit are cleared in lines 97 through 100 (R3 is an extension of the timer).

At lines 101 through 107 we wait for one complete cycle at the TO input, with the timer/counter measuring that period, then return to line 80, where the CALC subroutine is called.

The CALC subroutine, starting at line 113, begins by initializing the word to send out

(R7/R6) to zero, clearing the carry bit, checking to see if R3 indicates a time above one second, returning to line 81 if it does. Otherwise the program continues at line 26, where the program checks to see if the input frequency is beyond full scale (timer reading above 00 12 88 hex). If it is, R7/R6 is loaded with 12 88 hex (full scale of decimal 1,000). This value was chosen because it is sufficiently far from zero scale that it is easily discerned from zero scale on the display.

If the result is not to be full scale or zero scale, the program continues at line 140 with a shift and subtract divide routine. The dividend would be 1,000,000 (decimal) to convert back to frequency in Hertz (period measurements is in microseconds), but that would provide a maximum count of 200 at 200Hz, only one fifth of the full scale desired of 1,000. So we made the dividend to be 5,000,000 decimal, or 4C 4B 40 hex.

This algorithm is found in lines 156 through 192 and works as follows:

1. Clear a counter.
2. Rotate dividend until the first one is in the second MSB position. Since a code of 4C has already provided that, no shifting is necessary.
3. Rotate the divisor (the period in microseconds in this case) left until the first one is in the second MSB position, but the first byte is LESS THAN the first byte of the dividend. Increment the counter each time the divisor is rotated.
4. Initialize a counter to zero.
5. Rotate the quotient (answer) and dividend one bit left.
6. If first byte of quotient is smaller than the first byte of the quotient, jump to step 8.
7. Add one to the quotient and subtract the divisor from the dividend.
8. Decrement the counter and go to step 5 if it is not zero.

Once the CALC subroutine is completed, the program calls SENDIT from line 81 and jumps, ultimately, to the selected routine.

Routine 3

PROG3, beginning at line 194, measures the input period four times, then calculates the

code to display that is the average of these four readings.

It starts by setting a counter for three readings, taking those three readings and storing them in memory, beginning at RAM address 20 hex, using register R0 as an index register.

At line 212 the program takes a fourth reading, then adds the three previous readings to it in lines 213 through 227; and divides the sum by four (rotates right twice) in lines 229 through 239. The word to send out is then calculated from line 240 and sent to the ACMD, after which the program then looks for and jumps to the selected routine.

Routine 4

PROG4 begins at line 243 and displays the average of the current and last three words sent out.

RAM space used is first initialized to zero and a new reading is taken and a new word is calculated and saved. At lines 264 through 284, the new word is added to the last three readings and the average calculated and stored in RAM locations 28 and 29 (hex), and the average word is sent out.

At line 286, the program reads for the program selected and jumps to line 254 if this routine is selected, otherwise it goes to line 28.

ROUTINE 5

PROG5 begins at line 293 and, very simply, send in sequence the codes for 1/8 through full scale in 1/8 scale steps, with 500ms between steps. It then steps down to zero scale in 1/8 scale steps, then returns to line 28.

Routine 6

PROG6 begins at line 314 and does the same as PROG5, but steps in 1/4 scale increments.

Routine 7

PROG7 loads the code for 3/8 scale into R7/R6, sends it, waits 500ms, changes r& for 5/8 scale, sends it, waits for 500ms, then repeats this sequence 9 more times (for a total of ten times), waits 500ms, then returns the output to zero scale and the program jumps to line 28.

Controlling air core meters with the 87C751 and SA5775 AN426

```

1 ; ACMD V3 DEMO TT.20
2 ; PROCESSOR: 87C751 7-29-89
3 ;
4 ;
5 ; The purpose of this program is to drive version 3 of the ACMD. (SA5775)
6 ; demonstration board. The PROGRAM SELECT switch is used to select from
7 ; a choice of four routines. Registers R7 and R6 contain the 10-bit word
8 ; that is sent to the SA5775.
9 ;
10 ; $MOD751
11 0000 11 ORG 0
12 ;
13 0000 B02E 13 SJMP START ; RESET VECTOR
14 ;
15 000B 15 ORG 00BH ; TIMER/COUNTER INTERRUPT ROUTINE
16 000B 0B 16 INC R3 ; INCREMENT R3 (3rd BYTE OF TIMER)
17 000C 740F 17 MOV A,#0FH ; TEST FOR TIME OUT (R3 > 0F)
18 000E 9B 18 SUBB A,R3 ; IF R3 > 0F, CARRY IS SET
19 000F 32 19 RETI
20 ;
21 0030 21 ORG 30H ; START OF PROGRAM
22 0030 7580FF 22 START: MOV P0,#0FFH ; SET PORTS HIGH
23 0033 7590FF 23 MOV P1,#0FFH
24 0036 75B0FF 24 MOV P3,#0FFH
25 0039 7F00 25 MOV R7,#0 ; CLEAR WORD TO SEND OUT
26 003B 7E00 26 MOV R6,#0
27 003D 20B6FD 27 W: JB P3.6,W ; WAIT FOR START BUTTON DEPRESS
28 0040 758BF0 28 READY: MOV R7,#LOW(0-10000) ; SET TIMER REGISTER
29 0043 758DD8 29 MOV RTH,#HIGH(0-10000) ; FOR 10ms TIME
30 0046 51D2 30 ACALL RPS ; READ PORT 3 FOR PROG SELECT
31 0048 90004C 31 MOV DPTR,#JMPTBL ; JMP ADDRESS TO DATA POINTER
32 004B 73 32 JMP @A+DPTR ; GOTO APPROPRIATE ROUTINE
33 004C 015C 33 JMPTBL: AJMP PROG0 ; RAMP UP AND BACK DOWN
34 004E 0168 34 AJMP PROG1 ; STEP UP/DOWN W/ start PRESS
35 0050 017A 35 AJMP PROG2 ; READ & DISPLAY SPEED
36 0052 2145 36 AJMP PROG3 ; DISPLAY AVERAGE OF 4 NEW READINGS
37 0054 2186 37 AJMP PROG4 ; DISPLAY AVERAGE OF LAST 4 READINGS
38 0056 21D3 38 AJMP PROG5 ; ADVANCE TO FULL SCALE AND BACK IN 45 DEGREE STEPS
39 0058 21F3 39 AJMP PROG6 ; ADVANCE TO FULL SCALE AND BACK IN 90 DEGREE STEPS
40 005A 4107 40 AJMP PROG7 ; ALTERNATE DISPLAY BETWEEN 3/8 AND 5/8 SCALE TEN TIMES
41 005C 41 41 PROG0:
42 ; This routine increases word sent at the selected step size (INCREMENT SELECT)
43 ; and delay time (DELAY), up to full scale, waits 500ms, then decreases the
44 ; word sent at the selected step size and delay times until zero scale is reached.
45 ;
46 005C 5128 46 ACALL SO ; SEND OUT INCREASING WORDS
47 005E 5121 47 ACALL FULLSC ; SET TO FULL SCALE
48 0060 51A5 48 ACALL DLY500 ; WAIT 500ms
49 0062 5152 49 ACALL SOD ; SEND OUT DECREASING WORDS
50 0064 511B 50 ACALL ZEROSC ; RESET TO ZERO SCALE
51 0066 0130 51 AJMP START ; GO TO BEGINNING OF PROGRAM
52 006B 52 PROG1:
53 ;
54 ; MANUAL INCREMENT/DECREMENT ROUTINE
55 ;
56 ; This routine increases or decreases the sent out word, depending upon
57 ; the setting of the UP/DOWN switch, by an amount set by the INCREMENT
58 ; SELECT switch. There is a wait of 200ms before again looking for
59 ; depression of the START/COUNT button to allow time to release this
60 ; button and switch bounce to settle. The program then looks to see which
61 ; routine is selected and goes to that routine.
62 ;
63 0068 30B50B 63 JNB P3.5,DCX ; GO AND COUNT DOWN IF SELECTED
64 006B 5130 64 ACALL UP ; INCREASE WORD
65 006D 51B5 65 DP1: ACALL SENDIT ; SEND THE WORD
66 006F 519D 66 ACALL DLY200 ; WAIT 200ms
67 0071 013D 67 AJMP W ; WAIT FOR COUNT BUTTON DEPRESS & SELECTED ROUTINE
68 0073 20B5F2 68 DCX: JB P3.5,PROG1 ; GO AND COUNT UP IF SELECTED
69 0076 515A 69 ACALL DOWN ; DECREASE WORD

```

Controlling air core meters with the 87C751 and SA5775

```

0078 80F3    70      SJMP DP1
007A         71      PROG2:
007A         72      ;
007A         73      ; READ TIME INPUT AND DISPLAY "SPEED"
007A         74      ;
007A         75      ; This routine measures the period of the square wave at the T0 input and
007A         76      ; sends out a word that is inversely proportional to 5 times that period,
007A         77      ; providing a display proportional to frequency.
007A         78      ;
007A 1182    79      ACALL MEAS      ;MEASURE THE INPUT PERIOD
007C 11C5    80      ACALL CALC      ;CALCULATE THE WORD TO SEND
007E 51B5    81      ACALL SENDIT    ;SEND OUT THE WORD
0080 0140    82      AJMP READY      ;
0082 C28C    83      MEAS: CLR TR      ;HALT TIMER
0084 C28D    84      CLR TF      ;CLEAR TIMER FLAG
0086 758B00  85      MOV R7L,#0      ;SET TIMER REGISTERS
0088 758D00  86      MOV R7H,#0      ;
008C 758A00  87      MOV TL,#0      ;SET TIMER
008F 758C00  88      MOV TH,#0      ;
0092 7B00    89      MOV R3,#0      ;CLEAR TIMER 3RD BYTE
0094 C3      90      CLR C
0095 D28C    91      SETB TR      ;START TIMER
0097 75A882  92      MOV IE,#82H      ;ENABLE TIMER INTERRUPT
009A 4021    93      W20: JC GZS      ;JUMP IF R3 > OF
009C 2097FB  94      JB P1.7,W20      ;WAIT FOR T0 INPUT LOW
009F 401C    95      W21: JC GZS      ;JUMP IF R3 > OF
00A1 3097FB  96      JNB P1.7,W21      ;WAIT FOR T0 INPUT HIGH
00A4 758A00  97      MOV TL,#0      ;RESET TIMER
00A7 758C00  98      MOV TH,#0      ;
00AA 7B00    99      MOV R3,#0      ;
00AC C3      100     CLR C      ;CLEAR CARRY/BORROW
00AD 4008    101     W22: JC HT      ;JUMP IF TIME UP (CARRY SET)
00AF 2097FB  102     JB P1.7,W22      ;WAIT FOR T0 LOW
00B2 4003    103     W23: JC HT      ;JUMP IF TIME UP (CARRY SET)
00B4 3097FB  104     JNB P1.7,W23      ;WAIT FOR T0 HIGH AGAIN
00B7 C28C    105     HT: CLR TR      ;HALT TIMER
00B9 75A800  106     MOV IE,#0      ;DISABLE ALL INTERRUPTS
00BC 22      107     RET
00BD 7B1F    108     GZS: MOV R3,#1FH      ;SET FOR ZERO SCALE
00BF 22      109     RET
00C0 7F03    110     GFS: MOV R7,#03      ;
00C2 7EE8    111     MOV R6,#0E8H      ;
00C4 22      112     RET
00C5         113     CALC:
00C5         114     ;
00C5         115     ; This subroutine calculates the 10-bit word to send as a function of what
00C5         116     ; is in R3, TH & TL. The 10-bit word is developed and left in registers
00C5         117     ; R7 and R6 for use by SENDIT subroutine.
00C5         118     ;
00C5 7F00    119     MOV R7,#0      ;INITIALIZE QUOTIENT
00C7 7E00    120     MOV R6,#0      ;
00C9 C3      121     CLR C      ;CLEAR CARRY/BORROW
00CA 740F    122     MOV A,#0FH      ;CHECK FOR ZERO SCALE
00CC 9B      123     SUBB A,R3
00CD 5001    124     JNC NZS      ;JUMP IF NOT ZERO SCALE
00CF 22      125     RET
00D0 E58A    126     NZS: MOV A,TL      ;CHECK FOR FULL SCALE
00D2 9488    127     SUBB A,#88H
00D4 E58C    128     MOV A,TH
00D6 9413    129     SUBB A,#13H
00D8 EB      130     MOV A,R3
00D9 9400    131     SUBB A,#0
00DB 40E3    132     JC GFS
00DD 752E4C  133     MOV R7,#2EH,#4CH      ;SET DIVIDEND TO 5,000,000
00DE 752F4B  134     MOV R7H,#2FH,#4BH      ;
00E3 753040  135     MOV R7H,#30H,#40H      ;
00E6 7C00    136     MOV R4,#0      ;CLEAR DIVIDE COUNTER
00E8 8B2B    137     MOV 2BH,R3      ;MOVE READING TO MEMORY (DIVISOR)
00EA 858C2C  138     MOV 2CH,TH

```

```

00ED 858A2D 139      MOV    2DH,TL
00F0 C3 140      ROTL:  CLR    C                ;BRING DIVISOR BE JUST LESS THAN DIVIDEND
00F1 E52E 141      MOV    A,2EH
00F3 952B 142      SUBB   A,2BH                "CHECK" VALUE FOR THE THIRTY UNIT GAIN
00F5 4014 143      JC      DIV24                ;JUMP IF SHIFTING WOULD MAKE DIVISOR > DIVIDEND
00F7 6012 144      JZ      DIV24                ;JUMP IF DIVISOR & DIVIDEND MS BYTES EQUAL BEFORE SHIFT
00F9 E52D 145      MOV    A,2DH                ;SHIFT DIVISOR TO LEFT
00FB 33 146      RLC    A
00FC F52D 147      MOV    2DH,A
00FE E52C 148      MOV    A,2CH                ;CHECK THE THIRTY UNIT GAIN
0100 33 149      RLC    A                ;CARRY THE THIRTY UNIT GAIN
0101 F52C 150      MOV    2CH,A                ;CARRY THE THIRTY UNIT GAIN
0103 E52B 151      MOV    A,2BH
0105 33 152      RLC    A                ;CARRY THE THIRTY UNIT GAIN
0106 F52B 153      MOV    2BH,A                ;CARRY THE THIRTY UNIT GAIN
0108 0C 154      INC     R4                ;INCREASE THE THIRTY UNIT GAIN
0109 80B5 155      SJMP   ROTL                ;JUMP TO ROTL
010B C3 156      DIV24: CLR    C
010C EE 157      MOV    A,R6                ;ROTATE QUOTIENT LEFT
010D 33 158      RLC    A                ;CARRY THE THIRTY UNIT GAIN
010E FE 159      MOV    R6,A
010F EF 160      MOV    A,R7                ;CARRY THE THIRTY UNIT GAIN
0110 33 161      RLC    A                ;CARRY THE THIRTY UNIT GAIN
0111 FF 162      MOV    R7,A                ;CARRY THE THIRTY UNIT GAIN
0112 C3 163      CLR    C                ;ROTATE DIVIDEND LEFT
0113 E530 164      MOV    A,30H                ;CARRY THE THIRTY UNIT GAIN
0115 33 165      RLC    A                ;CARRY THE THIRTY UNIT GAIN
0116 F530 166      MOV    30H,A                ;CARRY THE THIRTY UNIT GAIN
0118 E52F 167      MOV    A,2FH
011A 33 168      RLC    A                ;CARRY THE THIRTY UNIT GAIN
011B F52F 169      MOV    2FH,A                ;CARRY THE THIRTY UNIT GAIN
011D E52E 170      MOV    A,2EH                ;CARRY THE THIRTY UNIT GAIN
011F 33 171      RLC    A                ;CARRY THE THIRTY UNIT GAIN
0120 F52E 172      MOV    2EH,A                ;CARRY THE THIRTY UNIT GAIN
0122 C3 173      CLR    C                ;TEST SUBTRACT MOST SIGNIFICANT BYTES
0123 952B 174      SUBB   A,2BH                ;CARRY THE THIRTY UNIT GAIN
0125 401B 175      JC      ZERO                ;JUMP IF QUOTIENT MS BYTE < DIVISOR MS BYTE
0127 7401 176      MOV    A,#1                ;ADD 1 TO QUOTIENT
0129 2E 177      ADD     A,R6                ;CARRY THE THIRTY UNIT GAIN
012A FE 178      MOV    R6,A
012B EF 179      MOV    A,R7                ;CARRY THE THIRTY UNIT GAIN
012C 3400 180      ADDC   A,#0                ;CARRY THE THIRTY UNIT GAIN
012E FF 181      MOV    R7,A
012F C3 182      CLR    C                ;SUBTRACT DIVISOR FROM DIVIDEND
0130 E530 183      MOV    A,30H
0132 952D 184      SUBB   A,2DH                ;CARRY THE THIRTY UNIT GAIN
0134 F530 185      MOV    30H,A                ;CARRY THE THIRTY UNIT GAIN
0136 E52F 186      MOV    A,2FH
0138 952C 187      SUBB   A,2CH                ;CARRY THE THIRTY UNIT GAIN
013A F52F 188      MOV    2FH,A                ;CARRY THE THIRTY UNIT GAIN
013C E52E 189      MOV    A,2EH
013E 952B 190      SUBB   A,2BH                ;CARRY THE THIRTY UNIT GAIN
0140 F52E 191      MOV    2EH,A                ;CARRY THE THIRTY UNIT GAIN
0142 DCC7 192      ZERO:  DJNZ   R4,DIV24
0144 22 193      RET
0145 194      PROG3:
195      ;
196      ;      DISPLAY AVERAGE OF FOUR NEW READINGS
197      ;
198      ;      This routine reads the period of the T0 input four times, then displays the
199      ;      "speed" corresponding to the average of these four readings.
200      ;
0145 7903 201      MOV    R1,#3                ;SET FOR 3 READINGS
0147 7820 202      MOV    R0,#20H                ;SET INDEX REGISTER FOR BOTTOM
0149 1182 203      P30:  ACALL  MEAS                ;TAKE 3 READINGS AND SAVE THEM
014B EB 204      MOV    A,R3
014C F6 205      MOV    @R0,A                ;CARRY THE THIRTY UNIT GAIN
014D 08 206      INC     @R0                ;CARRY THE THIRTY UNIT GAIN
014E A68C 207      MOV    @R0,TH                ;CARRY THE THIRTY UNIT GAIN

```

```

0150 08      208      INC      R0
0151 A68A    209      MOV      @R0,TL
0153 08      210      INC      R0
0154 D9F3    211      DJNZ     R1,P30
0156 1182    212      ACALL    MEAS      ;TAKE A 4TH READING, LEAVING IN R3,TH,TL
0158 7828    213      MOV      R0,#28H      ;SET INDEX REGISTER FOR TOP
015A 7903    214      MOV      R1,#3      ;SET COUNTER TO ADD FIRST 3 READINGS TO LAST ONE
015C E58A    215      MOV      A,TL      ;ADD FIRST THREE READINGS TO THE LAST ONE
P31: 015E 26      216      ADD      A,@R0
015F F58A    217      MOV      TL,A
0161 18      218      DEC      R0
0162 E58C    219      MOV      A,TH
0164 36      220      ADDC     A,@R0
0165 F58C    221      MOV      TH,A
0167 18      222      DEC      R0
0168 EB      223      MOV      A,R3
0169 36      224      ADDC     A,@R0
016A FB      225      MOV      R3,A
016B 18      226      DEC      R0
016C D9EE    227      DJNZ     R1,P31
016E 7902    228      MOV      R1,#2
0170 EB      229      MOV      A,R3      ;DIVIDE BY 4 (ROTATE RIGHT TWICE) FOR AVERAGE
P32: 0171 C3      230      CLR      C
0172 13      231      RRC      A
0173 FB      232      MOV      R3,A
0174 E58C    233      MOV      A,TH
0176 13      234      RRC      A
0177 F58C    235      MOV      TH,A
0179 E58A    236      MOV      A,TL
017B 13      237      RRC      A
017C F58A    238      MOV      TL,A
017E D9F0    239      DJNZ     R1,P32
0180 11C5    240      ACALL    CALC      ;CALCULATE THE WORD
0182 51B5    241      ACALL    SENDIT    ;SEND OUT THE WORD
0184 0140    242      AJMP     READY      ;GO TO SELECTED ROUTINE
0186         243      PROG4:
          244      ;
          245      ;      DISPLAY AVERAGE OF LAST FOUR WORDS SENT OUT
          246      ;
          247      ;      This routine sends out the average of the last four readings sent out.
          248      ;
0186 7827    249      MOV      R0,#27H
0188 7600    250      MOV      @R0,#0
018A 18      251      DEC      R0
018B B81FFA  252      CJNE     R0,#1FH,P4
018E 7820    253      MOV      R0,#20H
0190 1182    254      P4A:    ACALL    MEAS      ;MEASURE PERIOD
0192 11C5    255      P40:    ACALL    CALC      ;CALCULATE THE CODE
0194 EF      256      MOV      A,R7
0195 F6      257      MOV      @R0,A      ;SAVE THE CODE
0196 08      258      INC      R0
0197 EE      259      MOV      A,R6
0198 F6      260      MOV      @R0,A
0199 752800  261      MOV      28H,#0      ;INITIALIZE THE WORD TO SEND
019C 752900  262      MOV      29H,#0
019F 7927    263      MOV      R1,#27H
01A1 E529    264      P41:    MOV      A,29H      ;ADD TOGETHER LAST 4 RESULTS
01A3 C3      265      CLR      C
01A4 27      266      ADD      A,@R1
01A5 F529    267      MOV      29H,A
01A7 E528    268      MOV      A,28H
01A9 19      269      DEC      R1
01AA 37      270      ADDC     A,@R1
01AB F528    271      MOV      28H,A
01AD 19      272      DEC      R1
01AE B91FF0  273      CJNE     R1,#1FH,P41
01B1 7902    274      MOV      R1,#2
01B3 C3      275      P42:    CLR      C
01B4 E528    276      MOV      A,28H

```


Controlling air core meters with the 87C751 and SA5775 AN426

```

01B6 13      277      RRC      A
01B7 F528    278      MOV      28H,A
01B9 E529    279      MOV      A,29H
01BB 13      280      RRC      A
01BC F529    281      MOV      29H,A
01BE D9F3    282      DJNZ     R1,P42
01C0 AF28    283      MOV      R7,28H
01C2 AE29    284      MOV      R6,29H
01C4 51B5    285      ACALL    SENDIT      ;SEND OUT THE WORD
01C6 51D2    286      ACALL    RPS      ;READ PROGRAM SELECT
01C8 B40806  287      CJNE     A,#8,N4      ;JUMP TO N4 (& "READY") IF PROGRAM 4 NOT SELECTED
01CB 08      288      INC      R0
01CC B828C1  289      CJNE     R0,#28H,P40      ;GOTO P40 IF R0 NOT 28 (HEX)A
01CF 80BD    290      SJMP     P4A
01D1 0140    291      N4:      AJMP     READY
                292      ;
                293      PROG5:
                294      ;
                295      ; This routine advances the display in 45 degree steps to full scale, then steps down
                296      ; to zero in 45 degree steps. There is a 500ms delay between steps.
                297      ;
01D3 7F00    298      MOV      R7,#0
01D5 7E7F    299      P5:      MOV      R6,#07FH
01D7 51B1    300      ACALL    SD500      ;SEND THE WORD AND WAIT 500ms
01D9 7EFF    301      MOV      R6,#0FFH
01DB 51B1    302      ACALL    SD500      ;SEND THE WORD AND WAIT 500ms
01DD 0F      303      INC      R7
01DE BF04F4  304      CJNE     R7,#4,P5
01E1 7F03    305      MOV      R7,#3
01E3 7EFF    306      LP5:      MOV      R6,#0FFH
01E5 51B1    307      ACALL    SD500      ;SEND THE WORD AND WAIT 500ms
01E7 7E7F    308      MOV      R6,#7FH
01E9 51B1    309      ACALL    SD500
01EB 1F      310      DEC      R7
01EC BFFFF4  311      CJNE     R7,#0FFH,LP5
01EF 511B    312      ACALL    ZEROSC      ;RETURN TO ZERO
01F1 013D    313      AJMP     W      ;WAIT FOR KEY PRESS
01F3         314      PROG6:
                315      ;
                316      ; This routine advances the display in 90 degree steps to full scale, then steps down
                317      ; to zero in 90 degree steps. There is a 500ms delay between steps.
                318      ;
01F3 7EFF    319      MOV      R6,#0FFH
01F5 7F00    320      MOV      R7,#0
01F7 51B1    321      LP6:      ACALL    SD500      ;SEND THE WORD AND WAIT 500ms
01F9 0F      322      INC      R7
01FA BF04FA  323      CJNE     R7,#4,LP6
01FD 1F      324      LP6A:      DEC      R7
01FE 51B1    325      ACALL    SD500      ;SEND THE WORD AND WAIT 500ms
0200 BF00FA  326      CJNE     R7,#0,LP6A
0203 511B    327      ACALL    ZEROSC      ;RETURN TO ZERO
0205 013D    328      AJMP     W      ;WAIT FOR KEY PRESS
0207         329      PROG7:
                330      ;
                331      ; This routine alternates between 3/8 and 5/8 scale ten times with 300ms delay
                332      ; between steps, then waits 500ms before returning display to zero scale.
                333      ;
0207 7A0A    334      MOV      R2,#10      ;SET COUNTER
0209 7E7F    335      PR7:      MOV      R6,#07FH
020B 7F01    336      MOV      R7,#1
020D 51AD    337      ACALL    SD300      ;SEND OUT THE WORD AND WAIT 300ms
020F 7F02    338      MOV      R7,#2
0211 51AD    339      ACALL    SD300      ;SEND OUT THE WORD AND WAIT 300ms
0213 DAF4    340      DJNZ     R2,PR7      ;DO IT 10 TIMES
0215 51A5    341      ACALL    DLY500      ;WAIT 500ms
0217 511B    342      ACALL    ZEROSC      ;RESET TO ZERO SCALE
0219 0130    343      AJMP     START      ;LOOK FOR VALID PROGRAM
                344      ;
                345      ;

```


Controlling air core meters with the 87C751 and SA5775

```

346 ; SUBROUTINES
347 ;
348 ;
021B 7F00 349 ZEROSC: MOV R7,#0 ;RESET METER TO ZERO SCALE
021D 7E00 350 MOV R6,#0
021F 4125 351 AJMP RST ;
0221 7F03 352 FULLSC: MOV R7,#03H ;SET METER TO FULL SCALE
0223 7EFF 353 MOV R6,#0FFH
0225 51B5 354 RST: ACALL SENDIT ;
0227 22 355 ;
0228 356 SO:
357 ;
358 ; This subroutine sends increasing 10-bit words in registers R7 & R6 to the ACMD.
359 ;
0228 51B5 360 ACALL SENDIT ;WRITE THE 10-BIT WORD TO ACMD
022A 5130 361 ACALL UPON ;INCREASE THE WORD VALUE
022C 30E2F9 362 JNB ACC.2,SO ;JUMP IF BIT 2 NOT SET
022F 22 363 RET
0230 364 UP:
365 ;
366 ; This subroutine waits for a period of time = 10ms X DELAY read un, then
367 ; increases the 10-bit word by the INCREMENT SELECT amount.
368 ;
0230 E590 369 MOV A,P1 ;READ DELAY
0232 F4 370 CPL A ;COMPLEMENT ACC
0233 540F 371 ANL A,#0FH ;MASK OFF UPPER 4 BITS
0235 23 372 RL A
0236 23 373 RL A
0237 F9 374 MOV R1,A
0238 B90002 375 CJNE R1,#0,D10 ;JUMP IF DELAY SET FOR ZERO
023B 8006 376 SJMP NODLY
023D 7B01 377 D10: MOV R3,#1 ;SET FOR 1 x 10ms DELAY
023F 5195 378 D10A: ACALL DLY10MS ;DELAY 10MS x DELAY
0241 D9FC 379 DJNZ R1,D10A
0243 E5B0 380 NODLY: MOV A,P3 ;READ INCREMENT SELECT
0245 F4 381 CPL A ;COMPLEMENT ACC
0246 5403 382 ANL A,#3 ;MASK OFF UPPER 6 BITS
0248 23 383 RL A
0249 23 384 RL A
024A 23 385 RL A
024B 04 386 INC A
024C 2E 387 ADD A,R6 ;ADD INCREMENT TO R6
024D FE 388 MOV R6,A ;SAVE IT
024E E4 389 CLR A
024F 3F 390 ADDC A,R7 ;ADD CARRY TO R7
0250 FF 391 MOV R7,A ;SAVE IT
0251 22 392 RET
0252 393 SOD:
394 ;
395 ; This subroutine sends out decreasing words at the rate set by DELAY and
396 ; step size determined by INCREMENT SELECT.
397 ;
0252 51B5 398 ACALL SENDIT ;SEND OUT THE PRESENT WORD
0254 515A 399 ACALL DOWN ;DECREASE THE WORD
0256 50FA 400 JNC SOD ;DO IT AGAIN IF CARRY NOT SET
0258 411B 401 AJMP ZEROSC
025A 402 DOWN:
403 ;
404 ; Waits for 10ms x DELAY pot setting, then sends out decreasing values of words
405 ; in step sizes of 8 x INCREMENT SELECT + 1.
406 ;
025A E590 407 MOV A,P1 ;READ DELAY
025C F4 408 CPL A ;COMPLEMENT ACC
025D 540F 409 ANL A,#0FH ;MASK OFF UPPER FOUR BITS
025F 23 410 RL A
0260 23 411 RL A
0261 F9 412 MOV R1,A ;SAVE DELAY
0262 B90002 413 CJNE R1,#0,D10S ;JUMP IF DELAY SET FOR ZERO
0265 8004 414 SJMP NDD

```

Controlling air core meters with the 87C751 and SA5775

0267 5195	415	D10S:	ACALL DLY10MS	;DELAY 10ms x (DELAY +1)	
0269 D9FC	416		DJNZ R1,D10S		
026B E5B0	417	NDD:	MOV A,P3	;READ INCREMENT SELECT	
026D F4	418		CPL A	;COMPLEMENT ACC	
026E 5403	419		ANL A,#3	;MASK OFF UPPER 6 BITS	
0270 23	420		RL A	;MULTIPLY BY 8	
0271 23	421		RL A		
0272 23	422		RL A		
0273 04	423		INC A	;INSURE MINIMUM STEP	
0274 C3	424		CLR C	;CLEAR CARRY FOR SUBTRACTION	
0275 CE	425		XCH A,R6		
0276 9E	426		SUBB A,R6	;SUBTRACT INCREMENT FROM R6	
0277 CE	427		XCH A,R6	;SAVE IT	
0278 E4	428		CLR A	;CLEAR ACCUM FOR SUBTRACTION	
0279 CF	429		XCH A,R7		
027A 9F	430		SUBB A,R7	;SUBTRACT BORROW FROM R7	
027B 5403	431		ANL A,#3	;INSURE MAXIMUM WORD	
027D CF	432		XCH A,R7	;SAVE IT	
027E 22	433		RET		
027F 00	434	DELAY:	NOP	;30s DELAY	
0280 22	435	RET			
0281	436	DMS10:			
	437				
	438				
	439				
	440				
	441				
0281 758AF0	442		MOV TL,#LOW,(0-10000)	;LOAD TIMER FOR 10ms DELAY	
0284 758CD8	443		MOV TH,#HIGH(0-10000)		
0287 C28D	444		CLR TF	;CLEAR TIMER FLAG	
0289 D28C	445		SETB TR	;START TIMER	
028B 308DFD	446	MS10W:	JNB TF,MS10W	;WAIT FOR TIMER FLAG TO BE SET	
028E C28D	447		CLR TF	;CLEAR TIMER FLAG	
0290 DBF9	448		DJNZ R3,MS10W	;WAIT RS x 10ms	
0292 C28C	449		CLR TR	;STOP TIMER	
0294 22	450		RET		
	451				
0295 7B01	452	DLY10MS:	MOV R3,#1	;SET R3 FOR 10ms WAIT	
0297 80EB	453		SJMP DMS10	;WAIT 10ms	
	454				
0299 7B0A	455	DLY100:	MOV R3,#10	;SET R3 FOR 100ms WAIT	
029B 80E4	456		SJMP DMS10	;WAIT 100ms	
	457				
029D 7B14	458	DLY200:	MOV R3,#20	;SET R3 FOR 200ms WAIT	
029F 80E0	459		SJMP DMS10	;WAIT 200ms	
	460				
02A1 7B1E	461	DLY300:	MOV R3,#30	;SET R3 FOR 300ms WAIT	
02A3 80DC	462		SJMP DMS10	;WAIT 300ms	
	463				
02A5 7B32	464	DLY500:	MOV R3,#50	;SET R3 FOR 500ms WAIT	
02A7 80D8	465		SJMP DMS10	;WAIT 500ms	
	466				
02A9 51B5	467	SD200:	ACALL SENDIT	;SEND THE WORD	
02AB 80F0	468		SJMP DLY200	;WAIT 200ms	
	469				
02AD 51B5	470	SD300:	ACALL SENDIT	;SEND THE WORD	
02AF 80F0	471		SJMP DLY300	;WAIT 200ms	
	472				
02B1 51B5	473	SD500:	ACALL SENDIT	;SEND THE WORD	
02B3 80F0	474		SJMP DLY500	;WAIT 500ms	
	475				
02B5	476	SENDIT:			
	477				
	478				
	479				
	480				
02B5 D282	481		SETB P0.2	;SET CS HIGH	
02B7 7902	482		MOV R1,#02	;SET COUNTER FOR 2 BITS OF R7	
02B9 EF	483		MOV A,R7	;MOVE R7 TO A FOR SEND OUT	

Controlling air core meters with the 87C751 and SA5775 AN426

```

02BA 13      484      RRC      A          ;ALIGN R7 FOR SEND OUT
02BB 13      485      RRC      A
02BC 13      486      RRC      A
02BD 51C7    487      ACALL    SEND1      ;SEND OUT UPPER TWO BITS
02BF 7908    488      MOV      R1,#8      ;SET COUNTER FOR R6 SEND OUT
02C1 EE      489      MOV      A,R6      ;MOVE R6 TO ACCUM
02C2 51C7    490      ACALL    SEND1      ;SEND OUT LOWER 8 BITS
02C4 C282    491      CLR      P0.2      ;LOAD ACMD
02C6 22      492      RET
02C7          493      SEND1:
02C7          494      ;
02C7          495      ; This subroutine sends [R1] number of bits of the accumulator, starting
02C7          496      ; with the MSB over the IIC port.
02C7          497      ; Accumulator, R0 and R1 are destroyed.
02C7          498      ;
02C7 33      499      RLC      A          ;ROTATE BIT TO CARRY
02C8 9281    500      MOV      P0.1,C      ;MOVE CARRY TO DATA OUT
02CA C280    501      CLR      P0.0      ;CLOCK LOW
02CC 00      502      NOP
02CD D280    503      SETB     P0.0      ;CLOCK HIGH
02CF D9F6    504      DJNZ     R1,SEND1      ;SEND NEXT BIT TILL DONE
02D1 22      505      RET
02D1          506      ;
02D2 E5B0    507      RPS:     MOV      A,P3      ;READ PORT 3 FOR PROGRAM SELECT
02D4 F4      508      CPL      A          ;COMPLEMENT ACC
02D5 03      509      RR      A          ;ROTATE TO LSB's & MULT BY 2
02D6 540E    510      ANL      A,#0EH      ;MASK FOR PROGRAM SELECT *2
02D8 DD      511      RET
02D8          512      END

```

ASSEMBLY COMPLETE, 0 ERRORS FOUND

Controlling air core meters with the 87C751 and SA5775

Symbol	Mode	Address	Value / Comment
ACC	D	ADDR	00E0H
CALC	C	ADDR	00C5H
D10	C	ADDR	023DH
D10A	C	ADDR	023FH
D10S	C	ADDR	0267H
DCX	C	ADDR	0073H
DELAY	C	ADDR	027FH
DIV24	C	ADDR	010BH
DLY100	C	ADDR	0299H
DLY10MS	C	ADDR	0295H
DLY200	C	ADDR	029DH
DLY300	C	ADDR	02A1H
DLY500	C	ADDR	02A5H
DMS10	C	ADDR	0281H
DOWN	C	ADDR	025AH
DP1	C	ADDR	006DH
FULLSC	C	ADDR	0221H
GFS	C	ADDR	00C0H
GZS	C	ADDR	00BDH
HT	C	ADDR	00B7H
IE	D	ADDR	00A8H
JMPTBL	C	ADDR	004CH
LP5	C	ADDR	01E3H
LP6	C	ADDR	01F7H
LP6A	C	ADDR	01FDH
MEAS	C	ADDR	0082H
MS10W	C	ADDR	028BH
N4	C	ADDR	01D1H
NDD	C	ADDR	026BH
NODLY	C	ADDR	0243H
NZS	C	ADDR	00D0H
P0	D	ADDR	0080H
P1	D	ADDR	0090H
P3	D	ADDR	00B0H
P30	C	ADDR	0149H
P31	C	ADDR	015CH
P32	C	ADDR	0170H
P4	C	ADDR	0188H
P40	C	ADDR	0190H
P41	C	ADDR	01A1H
P42	C	ADDR	01B3H
P4A	C	ADDR	018EH
P5	C	ADDR	01D5H
PR7	C	ADDR	0209H
PROG0	C	ADDR	005CH
PROG1	C	ADDR	0068H
PROG2	C	ADDR	007AH
PROG3	C	ADDR	0145H
PROG4	C	ADDR	0186H
PROG5	C	ADDR	01D3H
PROG6	C	ADDR	01F3H
PROG7	C	ADDR	0207H
READY	C	ADDR	0040H
ROTL	C	ADDR	00F0H
RPS	C	ADDR	02D2H
RST	C	ADDR	0225H
RTH	D	ADDR	008DH
RTL	D	ADDR	008BH
SD200	C	ADDR	02A9H
SD300	C	ADDR	02ADH
SD500	C	ADDR	02B1H
SEND1	C	ADDR	02C7H
SENDIT	C	ADDR	02B5H
SO	C	ADDR	0228H
SOD	C	ADDR	0252H
START	C	ADDR	0030H
TF	B	ADDR	008DH
TH	D	ADDR	008CH
TL	D	ADDR	008AH
TR	B	ADDR	008CH
UP	C	ADDR	0230H
W	C	ADDR	003DH
W20	C	ADDR	009AH
W21	C	ADDR	009FH
W22	C	ADDR	00ADH
W23	C	ADDR	00B2H
ZERO	C	ADDR	0142H
ZEROSC	C	ADDR	021BH

Timer 1 in non-I²C applications of the 83/87C751/752 microcontrollers

AN427

The small package of the 83/87C751 and 83/87C752 microcontrollers includes two hardware-implemented timers: a 16-bit programmable timer, and a 10-bit fixed-rate timer. The programmable timer is available for the application program, and its operation is similar to the timer/counter of the 80C51 timer in mode 2. The fixed-rate timer, Timer 1, is typically employed as a watchdog timer for the I²C port communications and is not available for other uses.

In applications which do not take advantage of the I²C communications capability, the "silicon real estate" taken by Timer 1 is not necessarily lost—it can be used as a fixed-rate timer by the application. This timer can become useful in various cases, such as simple control applications that need a delay while doing some software activities in parallel, or generating a free-running repetitive waveform where the exact timing is not important. Another type of application is a watchdog timer prompting the user about unexpected operation of a system or its hardware, or resetting a program that "lost track."

TIMER 1 IMPLEMENTATION

Timer 1 is clocked once per machine cycle, which is the oscillator frequency divided by 12. The timer operation is enabled by setting the TIRUN bit (bit 4) in the I2CFG register. Writing a 0 into the TIRUN bit will stop and clear the timer. The timer is 10 bits wide, and when it reaches the terminal count of 1024 it carries out and sets the Timer 1 interrupt flag. An interrupt will occur if the Timer 1 interrupt is enabled by bit ETI (bit 4) of the Interrupt Enable (IE) register, and global interrupts are enabled by bit EA (bit 7) of the same IE register.

The vector address for the Timer 1 interrupt is 1B hex, and the interrupt service routine must start at this address. As with all 8051 family microcontrollers, only the Program Counter is pushed onto the stack upon interrupt (other registers that are used both by the interrupt

service routine and elsewhere must be explicitly saved). The Timer 1 interrupt flag is cleared by setting the CLRTI bit (bit 5) of the I2CFG register.

Note that when the I²C interface is not operating—SLAVEN, MASTRQ, and MASTER bits are all 0—the I²C hardware does not affect Timer 1. The SCL and SDA pins can be used as I/O pins, and the activity of these pins will not cause the timer to run, stop, or reset. Upon hardware reset of the microcontroller, the SLAVEN, MASTRQ, and MASTER bits are all reset, so the programmer does not have to worry about interaction between the SDA/SCL pins and the timer.

FIXED-RATE TIMER

The first programming example demonstrates simple fixed-rate operation. Upon reset, interrupts are enabled, and Timer 1 is started. A wait loop simulates the "application" program. The demonstration service routine simply sets a flag—in real life it could do something more useful, such as toggling an output pin. Note that the interrupt flag is cleared by setting CLRTI prior to returning from the service routine. Upon overflow, the timer will go on running, as the TIRUN bit is still set, so the interrupts will be spaced exactly 1024 clock cycles apart. If the service routine would toggle an output pin instead of setting a flag, its output would be a square wave with a period of 2048 cycles. For an application that demands a "one-shot" delay only, the service routine should clear the TIRUN bit in order to avoid subsequent interrupts.

WATCHDOG TIMER

A watchdog timer mechanism is typically applied in order to detect "abnormal" behavior of hardware. If the microcontroller operates in a very noisy environment, there might be a fear of the program "running wild" as a result of extremely violent EMI interference. In such

a case, a watchdog may take care to reset the microcontroller when the Timer 1 interrupt occurs. This could be applied in application programs with a repetitive nature—the software needs to reset the timer within 1024 machine cycles of the last reset.

In a system where something is supposed to occur regularly—for example, an interrupt for an external event—the watchdog is designed to "bite" when the hardware "sleeps" and the expected "something" does not happen for too long a time. The timer is allowed to run continuously, but when the expected event occurs, it resets the timer back to 0. When the timer is reset within 1024 cycles of the last reset, the application runs normally. If the event does not occur, the Timer 1 interrupt service routine will be activated to take care of the exception.

The second programming example demonstrates the watchdog. Upon Reset, the TIRUN bit, ETI, and global interrupts are enabled. The watchdog timer is reset and restarted by the small subroutine WdRst. The application is simulated by a loop of delays. Delay 1 is less than 1024 cycles, and when WdRst is called within Delay 1 intervals, no Timer 1 interrupt occurs. This represents normal operation of a "real life" application. When the delay from last reset is greater than 1024 cycles—representing a hardware exception—the interrupt will occur. The service routine for the watchdog is somewhat unusual, as it does not return to the program location where the interrupt occurred. Instead, the operation of the microcontroller is restarted at Reset. Upon entering the service routine, the interrupt is cleared and the timer is reset. Because execution does not return to the interrupted program with a RETI instruction, the interrupt pending flag is cleared by a call to a dummy subroutine XRETI. The program is restarted at Reset with a regular AJMP instruction. The stack pointer is explicitly reinitialized for the warm reset, so there is no danger of stack overflow upon repeated watchdog invocations.

Timer 1 in non-I²C applications of the 83/87C751/752 microcontrollers

AN427

```

1;
2;
3;
4;
5;
6;
7;
8;
9;
10;
11;
12;
13;
14;
15;
16;
17;
18;
19;
20;
21;
22;
23;
24;
25;
26;
27;
28;
29;
30;
31;
32;
33;
34;
35;
36;
37;
38;
39;
40;
41;
42;
43;
44;
45;
46;
47;
48;
49;
50;
51;
52;
53;
54;
55;
56;
57;
58;
59;
60;
61;
62;
63;
64;
65;
66;
67;
68;
69;
70;
71;
72;
73;
74;
75;
76;
77;
78;
79;
80;
81;
82;
83;
84;
85;
86;
87;
88;
89;
90;
91;
92;
93;
94;
95;
96;
97;
98;
99;
100;
101;
102;
103;
104;
105;
106;
107;
108;
109;
110;
111;
112;
113;
114;
115;
116;
117;
118;
119;
120;
121;
122;
123;
124;
125;
126;
127;
128;
129;
130;
131;
132;
133;
134;
135;
136;
137;
138;
139;
140;
141;
142;
143;
144;
145;
146;
147;
148;
149;
150;
151;
152;
153;
154;
155;
156;
157;
158;
159;
160;
161;
162;
163;
164;
165;
166;
167;
168;
169;
170;
171;
172;
173;
174;
175;
176;
177;
178;
179;
180;
181;
182;
183;
184;
185;
186;
187;
188;
189;
190;
191;
192;
193;
194;
195;
196;
197;
198;
199;
200;
201;
202;
203;
204;
205;
206;
207;
208;
209;
210;
211;
212;
213;
214;
215;
216;
217;
218;
219;
220;
221;
222;
223;
224;
225;
226;
227;
228;
229;
230;
231;
232;
233;
234;
235;
236;
237;
238;
239;
240;
241;
242;
243;
244;
245;
246;
247;
248;
249;
250;
251;
252;
253;
254;
255;
256;
257;
258;
259;
260;
261;
262;
263;
264;
265;
266;
267;
268;
269;
270;
271;
272;
273;
274;
275;
276;
277;
278;
279;
280;
281;
282;
283;
284;
285;
286;
287;
288;
289;
290;
291;
292;
293;
294;
295;
296;
297;
298;
299;
300;
301;
302;
303;
304;
305;
306;
307;
308;
309;
310;
311;
312;
313;
314;
315;
316;
317;
318;
319;
320;
321;
322;
323;
324;
325;
326;
327;
328;
329;
330;
331;
332;
333;
334;
335;
336;
337;
338;
339;
340;
341;
342;
343;
344;
345;
346;
347;
348;
349;
350;
351;
352;
353;
354;
355;
356;
357;
358;
359;
360;
361;
362;
363;
364;
365;
366;
367;
368;
369;
370;
371;
372;
373;
374;
375;
376;
377;
378;
379;
380;
381;
382;
383;
384;
385;
386;
387;
388;
389;
390;
391;
392;
393;
394;
395;
396;
397;
398;
399;
400;
401;
402;
403;
404;
405;
406;
407;
408;
409;
410;
411;
412;
413;
414;
415;
416;
417;
418;
419;
420;
421;
422;
423;
424;
425;
426;
427;
428;
429;
430;
431;
432;
433;
434;
435;
436;
437;
438;
439;
440;
441;
442;
443;
444;
445;
446;
447;
448;
449;
450;
451;
452;
453;
454;
455;
456;
457;
458;
459;
460;
461;
462;
463;
464;
465;
466;
467;
468;
469;
470;
471;
472;
473;
474;
475;
476;
477;
478;
479;
480;
481;
482;
483;
484;
485;
486;
487;
488;
489;
490;
491;
492;
493;
494;
495;
496;
497;
498;
499;
500;
501;
502;
503;
504;
505;
506;
507;
508;
509;
510;
511;
512;
513;
514;
515;
516;
517;
518;
519;
520;
521;
522;
523;
524;
525;
526;
527;
528;
529;
530;
531;
532;
533;
534;
535;
536;
537;
538;
539;
540;
541;
542;
543;
544;
545;
546;
547;
548;
549;
550;
551;
552;
553;
554;
555;
556;
557;
558;
559;
560;
561;
562;
563;
564;
565;
566;
567;
568;
569;
570;
571;
572;
573;
574;
575;
576;
577;
578;
579;
580;
581;
582;
583;
584;
585;
586;
587;
588;
589;
590;
591;
592;
593;
594;
595;
596;
597;
598;
599;
600;
601;
602;
603;
604;
605;
606;
607;
608;
609;
610;
611;
612;
613;
614;
615;
616;
617;
618;
619;
620;
621;
622;
623;
624;
625;
626;
627;
628;
629;
630;
631;
632;
633;
634;
635;
636;
637;
638;
639;
640;
641;
642;
643;
644;
645;
646;
647;
648;
649;
650;
651;
652;
653;
654;
655;
656;
657;
658;
659;
660;
661;
662;
663;
664;
665;
666;
667;
668;
669;
670;
671;
672;
673;
674;
675;
676;
677;
678;
679;
680;
681;
682;
683;
684;
685;
686;
687;
688;
689;
690;
691;
692;
693;
694;
695;
696;
697;
698;
699;
700;
701;
702;
703;
704;
705;
706;
707;
708;
709;
710;
711;
712;
713;
714;
715;
716;
717;
718;
719;
720;
721;
722;
723;
724;
725;
726;
727;
728;
729;
730;
731;
732;
733;
734;
735;
736;
737;
738;
739;
740;
741;
742;
743;
744;
745;
746;
747;
748;
749;
750;
751;
752;
753;
754;
755;
756;
757;
758;
759;
760;
761;
762;
763;
764;
765;
766;
767;
768;
769;
770;
771;
772;
773;
774;
775;
776;
777;
778;
779;
780;
781;
782;
783;
784;
785;
786;
787;
788;
789;
790;
791;
792;
793;
794;
795;
796;
797;
798;
799;
800;
801;
802;
803;
804;
805;
806;
807;
808;
809;
810;
811;
812;
813;
814;
815;
816;
817;
818;
819;
820;
821;
822;
823;
824;
825;
826;
827;
828;
829;
830;
831;
832;
833;
834;
835;
836;
837;
838;
839;
840;
8
```


Timer 1 in non-I²C applications of the 83/87C751/752 microcontrollers

AN427

					Time I Fixed Rate Timer	11/06/90	PAGE 2
TINT	II-08-90						
CLRTI.	B ADDR	00DDH	PREDEFINED				
EA .	B ADDR	00AFH	PREDEFINED				
ETI .	B ADDR	00ABH	PREDEFINED				
FLAGS.	D ADDR	0020H					
LOOP.	C ADDR	0026H					
RESET.	C ADDR	0020H					
TIMERI	C ADDR	001BH	NOT USED				
TIRUN.	B ADDR	00DCH	PREDEFINED				
TSTFLAG.	B ADDR	0000H					
WAIT .	C ADDR	0028H					

Timer I in non-I²C applications of the 83/87C751/752 microcontrollers

AN427

TIWD01 00100111

Timer I Watchdog

11-06-90 PAGE 1

```

1 ;*****
2
3 ;               Timer I Watchdog Timer Usage
4
5 ;This program demonstrates how to use Timer I on the 83C751 or 83C752
6 ;microcontrollers as a watchdog timer when the I2C port is not used.
7 ;Once started, Timer I must be cleared more often than once every 1024
8 ;machine cycles. If Timer I is allowed to overflow, a Timer I
9 ;interrupt will be generated. Thus, if global interrupts or the Timer
10 ;I interrupt are inhibited, the watchdog function will be disabled.
11 ;Also, if the watchdog interrupt occurs during another interrupt
12 ;service, it will be delayed until an RETI (return from interrupt)
13 ;instruction is executed. The I2C bus pins SCL and SDA may be used as
14 ;open drain outputs.
15
16 ;*****
17
18 $MOD751
19 $Title(Timer I Watchdog)
20 $Date(11-06-90)
21 $Debug
22
23          ORG      0
24          AJMP     Reset
25
26          ORG      1Bh          ;Timer I interrupt.
27 TimerI:   CLR      EA          ;Get here only if watchdog overflows.
28          CLR      TIRUN        ;Turn off Timer I.
29          SETB     CLRTI        ;Clear Timer I interrupt.
30          ACALL    XRETI        ;Force interrupt pending to clear.
31          AJMP     Reset        ;Do a warm start.
32 XRETI:    RETI
33
34 Reset:    MOV      SP,#7h      ;Initialize the stack pointer.
35
36 ;Note: it is important to force the stack pointer to a particular
37 ;starting value in this application because we may be re-starting
38 ;after a watchdog interrupt, with the stack in an unknown condition.
39
40          MOV      I2CFG,#0      ;Initialize I2CFG (set up CT0, CT1).
41          SETB     TIRUN        ;Enable Timer I run.
42          SETB     ETI          ;Enable Timer I interrupt.
43          SETB     EA          ;Enable interrupt system.
44
45
46 ;The following is a "dummy" main program to test the watchdog timer.
47
48 Loop:     ACALL    Delay1       ;Wait 901 machine cycles.
49          ACALL    WdRst        ;Reset Watchdog.
50          ACALL    Delay1       ;Wait 901 machine cycles.
51          ACALL    WdRst        ;Reset Watchdog.
52          ACALL    Delay1       ;Wait 901 + 4 for ACALL & prior RET.
53          ACALL    Delay2       ;Wait 108 + 2 for ACALL.
54          NOP        ;1016
55          NOP        ;1017
56          NOP        ;1018
57          NOP        ;1019
58          NOP        ;1020

```

Timer 1 in non-I²C applications of the 83/87C751/752 microcontrollers

AN427

TIME	PC	OP	DATA	COMMENT
0043	00	59	NOP	;1021
0044	00	60	NOP	;1022
0045	00	61	NOP	;1023
0046	00	62	NOP	;1024
0047	00	63	NOP	;1025
0048	00	64	NOP	;1026
0049	00	65	NOP	;1027
004A	00	66	NOP	;1028
004B	00	67	NOP	;1029
004C	0132	68	AJMP	Loop
		69		
004E	C2DC	70	WdRst: CLR	TIRUN
0050	D2DC	71	SETB	TIRUN
0052	22	72	RET	
		73		
0053	7480	74	Delay1: MOV	A, #128
0055	8002	75	SJMP	DLoop
0057	740F	76	Delay2: MOV	A, #15
0059	A3	77	DLoop: INC	DPTR
005A	A3	78	INC	DPTR
005B	14	79	DEC	A
005C	70FB	80	JNZ	Dloop
005E	22	81	RET	
		82		
		83	END	

ASSEMBLY COMPLETE, 0 ERRORS FOUND

Timer I in non-I²C applications of the 83/87C751/752 microcontrollers

AN427

TIWD	08-06-90	Timer I Watchdog	11-06-90	PAGE 31
CLRTI	B ADDR 00DDH	PREDEFINED	00	00 0000
DELAY1	C ADDR 0053H		00	00 0000
DELAY2	C ADDR 0057H		00	00 0000
DLOOP	C ADDR 0059H		00	00 0000
EA	B ADDR 00AFH	PREDEFINED	00	00 0000
ETI	B ADDR 00ABH	PREDEFINED	00	00 0000
I2CFG	D ADDR 00D8H	PREDEFINED	00	00 0000
LOOP	C ADDR 0032H		00	00 0000
RESET	C ADDR 0026H		00	00 0000
SP	D ADDR 0081H	PREDEFINED	00	00 0000
TIMERI	C ADDR 001BH	NOT USED	00	00 0000
TIRUN	B ADDR 00DCH	PREDEFINED	00	00 0000
WDRST	C ADDR 004EH		00	00 0000
XRETI	C ADDR 0025H		00	00 0000

Using the ADC and PWM of the 83C752/87C752 AN428

The Philips 83C752/87C752 is a single-chip control-oriented microcontroller. It is an 80C51 derivative, having the same basic architecture and powerful instruction set in a small 28-pin package. As "add-on" functions to a standard microcontroller, it offers an I²C small area network port, a five-channel multiplexed 8-bit analog-to-digital converter (ADC), and a pulse width modulation (PWM) output. The part is essentially the popular 8XC751 with the addition of the ADC and the PWM output.

There are many control applications for which this microcontroller can provide an almost-complete, low-cost solution. The A/D converter can monitor analog voltages of up to five sources. The PWM output can be used to generate an analog control voltage with the addition of a simple integrator circuit. Another potential use for the PWM output is as a driver of power-switching circuits for DC motor speed control.

The analog-to-digital converter has 8-bit resolution, and the conversion takes 40 machine cycles. A multiplexer selects one out of five input pins. The operation of the A/D

converter and the multiplexer is controlled by the ADCON register.

The repetition frequency of the PWM output pulses is determined by an 8-bit prescaler, programmed at register PWMP. The duty cycle of these pulses is determined by the contents of a compare register, PWM. In order to implement the pulse width modulator, the prescaler output drives an 8-bit counter. When the counter value matches the contents of the compare (PWM) register, the PWM output is set high, and when the counter reaches zero, the output is set low. The counter is modulo 255, so the duty cycle generated will be the PWM contents multiplied by 1/255.

The enclosed listing demonstrates usage of the A/D converter and the PWM. In order to communicate with the outside world, the program sends messages on a software-driven RS-232 port. The routines for sending messages via a software-controlled serial port can be quite useful, and for further discussion on those, please refer to Application Note 423: "Software Driven Serial Communication Routines for the 83C751 and 83C752 Microcontrollers."

Bit 5 of port 1 is used for the RS-232 communications, and in order to hook the microcontroller to a terminal, a buffer (e.g., MC1488) is needed. Timer 0 is used as the baud rate generator, where the timer value is defined by the symbol BaudVal. The programmed value will generate a 9600 baud rate with a 16MHz crystal.

The program, after initialization and sending a message to the terminal, scans all five A/D channel inputs and outputs the voltage read on the serial port, as a hexadecimal value. Circuit operation can be verified by comparing channel voltages with the reading at the terminal. The program follows with an infinite loop in which channel 0 of the A/D converter is read, and its value is used to program the PWM. A simple verification of the duty cycle can be done with a voltmeter: since it acts as an integrator, its reading will be proportional to the duty cycle. Reading of a voltmeter on the PWM output should be proportional to the channel 0 input voltage. If the analog reference voltage AV_{CC}, which is full-scale of the A/D measurement, is set to be exactly as V_{CC}, the PWM output will track channel 0 within about 20mV.

```

0000      ORG      0000H
0001      MOV     R0, #0
0002      MOV     R7, #5
0003      MOV     R2, #0
0004      MOV     R3, #0
0005      MOV     R4, #0
0006      MOV     R5, #0
0007      MOV     R6, #0
0008      MOV     R1, #0
0009      MOV     R0, #0
0010      MOV     R0, #0
0011      MOV     R0, #0
0012      MOV     R0, #0
0013      MOV     R0, #0
0014      MOV     R0, #0
0015      MOV     R0, #0
0016      MOV     R0, #0
0017      MOV     R0, #0
0018      MOV     R0, #0
0019      MOV     R0, #0
0020      MOV     R0, #0
0021      MOV     R0, #0
0022      MOV     R0, #0
0023      MOV     R0, #0
0024      MOV     R0, #0
0025      MOV     R0, #0
0026      MOV     R0, #0
0027      MOV     R0, #0
0028      MOV     R0, #0
0029      MOV     R0, #0
0030      MOV     R0, #0
0031      MOV     R0, #0
0032      MOV     R0, #0
0033      MOV     R0, #0
0034      MOV     R0, #0
0035      MOV     R0, #0
0036      MOV     R0, #0
0037      MOV     R0, #0
0038      MOV     R0, #0
0039      MOV     R0, #0
0040      MOV     R0, #0
0041      MOV     R0, #0
0042      MOV     R0, #0
0043      MOV     R0, #0
0044      MOV     R0, #0
0045      MOV     R0, #0
0046      MOV     R0, #0
0047      MOV     R0, #0
0048      MOV     R0, #0
0049      MOV     R0, #0
0050      MOV     R0, #0
0051      MOV     R0, #0
0052      MOV     R0, #0
0053      MOV     R0, #0
0054      MOV     R0, #0
0055      MOV     R0, #0
0056      MOV     R0, #0
0057      MOV     R0, #0
0058      MOV     R0, #0
0059      MOV     R0, #0
0060      MOV     R0, #0
0061      MOV     R0, #0
0062      MOV     R0, #0
0063      MOV     R0, #0
0064      MOV     R0, #0
0065      MOV     R0, #0
0066      MOV     R0, #0
0067      MOV     R0, #0
0068      MOV     R0, #0
0069      MOV     R0, #0
0070      MOV     R0, #0
0071      MOV     R0, #0
0072      MOV     R0, #0
0073      MOV     R0, #0
0074      MOV     R0, #0
0075      MOV     R0, #0
0076      MOV     R0, #0
0077      MOV     R0, #0
0078      MOV     R0, #0
0079      MOV     R0, #0
0080      MOV     R0, #0
0081      MOV     R0, #0
0082      MOV     R0, #0
0083      MOV     R0, #0
0084      MOV     R0, #0
0085      MOV     R0, #0
0086      MOV     R0, #0
0087      MOV     R0, #0
0088      MOV     R0, #0
0089      MOV     R0, #0
0090      MOV     R0, #0
0091      MOV     R0, #0
0092      MOV     R0, #0
0093      MOV     R0, #0
0094      MOV     R0, #0
0095      MOV     R0, #0
0096      MOV     R0, #0
0097      MOV     R0, #0
0098      MOV     R0, #0
0099      MOV     R0, #0
0100      MOV     R0, #0
0101      MOV     R0, #0
0102      MOV     R0, #0
0103      MOV     R0, #0
0104      MOV     R0, #0
0105      MOV     R0, #0
0106      MOV     R0, #0
0107      MOV     R0, #0
0108      MOV     R0, #0
0109      MOV     R0, #0
0110      MOV     R0, #0
0111      MOV     R0, #0
0112      MOV     R0, #0
0113      MOV     R0, #0
0114      MOV     R0, #0
0115      MOV     R0, #0
0116      MOV     R0, #0
0117      MOV     R0, #0
0118      MOV     R0, #0
0119      MOV     R0, #0
0120      MOV     R0, #0
0121      MOV     R0, #0
0122      MOV     R0, #0
0123      MOV     R0, #0
0124      MOV     R0, #0
0125      MOV     R0, #0
0126      MOV     R0, #0
0127      MOV     R0, #0
0128      MOV     R0, #0
0129      MOV     R0, #0
0130      MOV     R0, #0
0131      MOV     R0, #0
0132      MOV     R0, #0
0133      MOV     R0, #0
0134      MOV     R0, #0
0135      MOV     R0, #0
0136      MOV     R0, #0
0137      MOV     R0, #0
0138      MOV     R0, #0
0139      MOV     R0, #0
0140      MOV     R0, #0
0141      MOV     R0, #0
0142      MOV     R0, #0
0143      MOV     R0, #0
0144      MOV     R0, #0
0145      MOV     R0, #0
0146      MOV     R0, #0
0147      MOV     R0, #0
0148      MOV     R0, #0
0149      MOV     R0, #0
0150      MOV     R0, #0
0151      MOV     R0, #0
0152      MOV     R0, #0
0153      MOV     R0, #0
0154      MOV     R0, #0
0155      MOV     R0, #0
0156      MOV     R0, #0
0157      MOV     R0, #0
0158      MOV     R0, #0
0159      MOV     R0, #0
0160      MOV     R0, #0
0161      MOV     R0, #0
0162      MOV     R0, #0
0163      MOV     R0, #0
0164      MOV     R0, #0
0165      MOV     R0, #0
0166      MOV     R0, #0
0167      MOV     R0, #0
0168      MOV     R0, #0
0169      MOV     R0, #0
0170      MOV     R0, #0
0171      MOV     R0, #0
0172      MOV     R0, #0
0173      MOV     R0, #0
0174      MOV     R0, #0
0175      MOV     R0, #0
0176      MOV     R0, #0
0177      MOV     R0, #0
0178      MOV     R0, #0
0179      MOV     R0, #0
0180      MOV     R0, #0
0181      MOV     R0, #0
0182      MOV     R0, #0
0183      MOV     R0, #0
0184      MOV     R0, #0
0185      MOV     R0, #0
0186      MOV     R0, #0
0187      MOV     R0, #0
0188      MOV     R0, #0
0189      MOV     R0, #0
0190      MOV     R0, #0
0191      MOV     R0, #0
0192      MOV     R0, #0
0193      MOV     R0, #0
0194      MOV     R0, #0
0195      MOV     R0, #0
0196      MOV     R0, #0
0197      MOV     R0, #0
0198      MOV     R0, #0
0199      MOV     R0, #0
0200      MOV     R0, #0
0201      MOV     R0, #0
0202      MOV     R0, #0
0203      MOV     R0, #0
0204      MOV     R0, #0
0205      MOV     R0, #0
0206      MOV     R0, #0
0207      MOV     R0, #0
0208      MOV     R0, #0
0209      MOV     R0, #0
0210      MOV     R0, #0
0211      MOV     R0, #0
0212      MOV     R0, #0
0213      MOV     R0, #0
0214      MOV     R0, #0
0215      MOV     R0, #0
0216      MOV     R0, #0
0217      MOV     R0, #0
0218      MOV     R0, #0
0219      MOV     R0, #0
0220      MOV     R0, #0
0221      MOV     R0, #0
0222      MOV     R0, #0
0223      MOV     R0, #0
0224      MOV     R0, #0
0225      MOV     R0, #0
0226      MOV     R0, #0
0227      MOV     R0, #0
0228      MOV     R0, #0
0229      MOV     R0, #0
0230      MOV     R0, #0
0231      MOV     R0, #0
0232      MOV     R0, #0
0233      MOV     R0, #0
0234      MOV     R0, #0
0235      MOV     R0, #0
0236      MOV     R0, #0
0237      MOV     R0, #0
0238      MOV     R0, #0
0239      MOV     R0, #0
0240      MOV     R0, #0
0241      MOV     R0, #0
0242      MOV     R0, #0
0243      MOV     R0, #0
0244      MOV     R0, #0
0245      MOV     R0, #0
0246      MOV     R0, #0
0247      MOV     R0, #0
0248      MOV     R0, #0
0249      MOV     R0, #0
0250      MOV     R0, #0
0251      MOV     R0, #0
0252      MOV     R0, #0
0253      MOV     R0, #0
0254      MOV     R0, #0
0255      MOV     R0, #0
0256      MOV     R0, #0
0257      MOV     R0, #0
0258      MOV     R0, #0
0259      MOV     R0, #0
0260      MOV     R0, #0
0261      MOV     R0, #0
0262      MOV     R0, #0
0263      MOV     R0, #0
0264      MOV     R0, #0
0265      MOV     R0, #0
0266      MOV     R0, #0
0267      MOV     R0, #0
0268      MOV     R0, #0
0269      MOV     R0, #0
0270      MOV     R0, #0
0271      MOV     R0, #0
0272      MOV     R0, #0
0273      MOV     R0, #0
0274      MOV     R0, #0
0275      MOV     R0, #0
0276      MOV     R0, #0
0277      MOV     R0, #0
0278      MOV     R0, #0
0279      MOV     R0, #0
0280      MOV     R0, #0
0281      MOV     R0, #0
0282      MOV     R0, #0
0283      MOV     R0, #0
0284      MOV     R0, #0
0285      MOV     R0, #0
0286      MOV     R0, #0
0287      MOV     R0, #0
0288      MOV     R0, #0
0289      MOV     R0, #0
0290      MOV     R0, #0
0291      MOV     R0, #0
0292      MOV     R0, #0
0293      MOV     R0, #0
0294      MOV     R0, #0
0295      MOV     R0, #0
0296      MOV     R0, #0
0297      MOV     R0, #0
0298      MOV     R0, #0
0299      MOV     R0, #0
0300      MOV     R0, #0
0301      MOV     R0, #0
0302      MOV     R0, #0
0303      MOV     R0, #0
0304      MOV     R0, #0
0305      MOV     R0, #0
0306      MOV     R0, #0
0307      MOV     R0, #0
0308      MOV     R0, #0
0309      MOV     R0, #0
0310      MOV     R0, #0
0311      MOV     R0, #0
0312      MOV     R0, #0
0313      MOV     R0, #0
0314      MOV     R0, #0
0315      MOV     R0, #0
0316      MOV     R0, #0
0317      MOV     R0, #0
0318      MOV     R0, #0
0319      MOV     R0, #0
0320      MOV     R0, #0
0321      MOV     R0, #0
0322      MOV     R0, #0
0323      MOV     R0, #0
0324      MOV     R0, #0
0325      MOV     R0, #0
0326      MOV     R0, #0
0327      MOV     R0, #0
0328      MOV     R0, #0
0329      MOV     R0, #0
0330      MOV     R0, #0
0331      MOV     R0, #0
0332      MOV     R0, #0
0333      MOV     R0, #0
0334      MOV     R0, #0
0335      MOV     R0, #0
0336      MOV     R0, #0
0337      MOV     R0, #0
0338      MOV     R0, #0
0339      MOV     R0, #0
0340      MOV     R0, #0
0341      MOV     R0, #0
0342      MOV     R0, #0
0343      MOV     R0, #0
0344      MOV     R0, #0
0345      MOV     R0, #0
0346      MOV     R0, #0
0347      MOV     R0, #0
0348      MOV     R0, #0
0349      MOV     R0, #0
0350      MOV     R0, #0
0351      MOV     R0, #0
0352      MOV     R0, #0
0353      MOV     R0, #0
0354      MOV     R0, #0
0355      MOV     R0, #0
0356      MOV     R0, #0
0357      MOV     R0, #0
0358      MOV     R0, #0
0359      MOV     R0, #0
0360      MOV     R0, #0
0361      MOV     R0, #0
0362      MOV     R0, #0
0363      MOV     R0, #0
0364      MOV     R0, #0
0365      MOV     R0, #0
0366      MOV     R0, #0
0367      MOV     R0, #0
0368      MOV     R0, #0
0369      MOV     R0, #0
0370      MOV     R0, #0
0371      MOV     R0, #0
0372      MOV     R0, #0
0373      MOV     R0, #0
0374      MOV     R0, #0
0375      MOV     R0, #0
0376      MOV     R0, #0
0377      MOV     R0, #0
0378      MOV     R0, #0
0379      MOV     R0, #0
0380      MOV     R0, #0
0381      MOV     R0, #0
0382      MOV     R0, #0
0383      MOV     R0, #0
0384      MOV     R0, #0
0385      MOV     R0, #0
0386      MOV     R0, #0
0387      MOV     R0, #0
0388      MOV     R0, #0
0389      MOV     R0, #0
0390      MOV     R0, #0
0391      MOV     R0, #0
0392      MOV     R0, #0
0393      MOV     R0, #0
0394      MOV     R0, #0
0395      MOV     R0, #0
0396      MOV     R0, #0
0397      MOV     R0, #0
0398      MOV     R0, #0
0399      MOV     R0, #0
0400      MOV     R0, #0
0401      MOV     R0, #0
0402      MOV     R0, #0
0403      MOV     R0, #0
0404      MOV     R0, #0
0405      MOV     R0, #0
0406      MOV     R0, #0
0407      MOV     R0, #0
0408      MOV     R0, #0
0409      MOV     R0, #0
0410      MOV     R0, #0
0411      MOV     R0, #0
0412      MOV     R0, #0
0413      MOV     R0, #0
0414      MOV     R0, #0
0415      MOV     R0, #0
0416      MOV     R0, #0
0417      MOV     R0, #0
0418      MOV     R0, #0
0419      MOV     R0, #0
0420      MOV     R0, #0
0421      MOV     R0, #0
0422      MOV     R0, #0
0423      MOV     R0, #0
0424      MOV     R0, #0
0425      MOV     R0, #0
0426      MOV     R0, #0
0427      MOV     R0, #0
0428      MOV     R0, #0
0429      MOV     R0, #0
0430      MOV     R0, #0
0431      MOV     R0, #0
0432      MOV     R0, #0
0433      MOV     R0, #0
0434      MOV     R0, #0
0435      MOV     R0, #0
0436      MOV     R0, #0
0437      MOV     R0, #0
0438      MOV     R0, #0
0439      MOV     R0, #0
0440      MOV     R0, #0
0441      MOV     R0, #0
0442      MOV     R0, #0
0443      MOV     R0, #0
0444      MOV     R0, #0
0445      MOV     R0, #0
0446      MOV     R0, #0
0447      MOV     R0, #0
0448      MOV     R0, #0
0449      MOV     R0, #0
0450      MOV     R0, #0
0451      MOV     R0, #0
0452      MOV     R0, #0
0453      MOV     R0, #0
0454      MOV     R0, #0
0455      MOV     R0, #0
0456      MOV     R0, #0
0457      MOV     R0, #0
0458      MOV     R0, #0
0459      MOV     R0, #0
0460      MOV     R0, #0
0461      MOV     R0, #0
0462      MOV     R0, #0
0463      MOV     R0, #0
0464      MOV     R0, #0
0465      MOV     R0, #0
0466      MOV     R0, #0
0467      MOV     R0, #0
0468      MOV     R0, #0
0469      MOV     R0, #0
0470      MOV     R0, #0
0471      MOV     R0, #0
0472      MOV     R0, #0
0473      MOV     R0, #0
0474      MOV     R0, #0
0475      MOV     R0, #0
0476      MOV     R0, #0
0477      MOV     R0, #0
0478      MOV     R0, #0
0479      MOV     R0, #0
0480      MOV     R0, #0
0481      MOV     R0, #0
0482      MOV     R0, #0
0483      MOV     R0, #0
0484      MOV     R0, #0
0485      MOV     R0, #0
0486      MOV     R0, #0
0487      MOV     R0, #0
0488      MOV     R0, #0
0489      MOV     R0, #0
0490      MOV     R0, #0
0491      MOV     R0, #0
0492      MOV     R0, #0
0493      MOV     R0, #0
0494      MOV     R0, #0
0495      MOV     R0, #0
0496      MOV     R0, #0
0497      MOV     R0, #0
0498      MOV     R0, #0
0499      MOV     R0, #0
0500      MOV     R0, #0
0501      MOV     R0, #0
0502      MOV     R0, #0
0503      MOV     R0, #0
0504      MOV     R0, #0
0505      MOV     R0, #0
0506      MOV     R0, #0
0507      MOV     R0, #0
0508      MOV     R0, #0
0509      MOV     R0, #0
0510      MOV     R0, #0
0511      MOV     R0, #0
0512      MOV     R0, #0
0513      MOV     R0, #0
0514      MOV     R0, #0
0515      MOV     R0, #0
0516      MOV     R0, #0
0517      MOV     R0, #0
0518      MOV     R0, #0
0519      MOV     R0, #0
0520      MOV     R0, #0
0521      MOV     R0, #0
0522      MOV     R0, #0
0523      MOV     R0, #0
0524      MOV     R0, #0
0525      MOV     R0, #0
0526      MOV     R0, #0
0527      MOV     R0, #0
0528      MOV     R0, #0
0529      MOV     R0, #0
0530      MOV     R0, #0
0531      MOV     R0, #0
0532      MOV     R0, #0
0533      MOV     R0, #0
0534      MOV     R0, #0
0535      MOV     R0, #0
0536      MOV     R0, #0
0537      MOV     R0, #0
0538      MOV     R0, #0
0539      MOV     R0, #0
0540      MOV     R0, #0
0541      MOV     R0, #0
0542      MOV     R0, #0
0543      MOV     R0, #0
0544      MOV     R0, #0
0545      MOV     R0, #0
0546      MOV     R0, #0
0547      MOV     R0, #0
0548      MOV     R0, #0
0549      MOV     R0, #0
0550      MOV     R0, #0
0551      MOV     R0, #0
0552      MOV     R0, #0
0553      MOV     R0, #0
0554      MOV     R0, #0
0555      MOV     R0, #0
0556      MOV     R0, #0
0557      MOV     R0, #0
0558      MOV     R0, #0
0559      MOV     R0, #0
0560      MOV     R0, #0
0561      MOV     R0, #0
0562      MOV     R0, #0
0563      MOV     R0, #0
0564      MOV     R0, #0
0565      MOV     R0, #0
0566      MOV     R0, #0
0567      MOV     R0, #0
0568      MOV     R0, #0
0569      MOV     R0, #0
0570      MOV     R0, #0
0571      MOV     R0, #0
0572      MOV     R0, #0
0573      MOV     R0, #0
0574      MOV     R0, #0
0575      MOV     R0, #0
0576      MOV     R0, #0
0577      MOV     R0, #0
0578      MOV     R0, #0
0579      MOV     R0, #0
0580      MOV     R0, #0
0581      MOV     R0, #0
0582      MOV     R0, #0
0583      MOV     R0, #0
0584      MOV     R0, #0
0585      MOV     R0, #0
0586      MOV     R0, #0
0587      MOV     R0, #0
0588      MOV     R0, #0
0589      MOV     R0, #0
0590      MOV     R0, #0
0591      MOV     R0, #0
0592      MOV     R0, #0
0593      MOV     R0, #0
0594      MOV     R0, #0
0595      MOV     R0, #0
0596      MOV     R0, #0
0597      MOV     R0, #0
0598      MOV     R0, #0
0599      MOV     R0, #0
0600      MOV     R0, #0
0601      MOV     R0, #0
0602      MOV     R0, #0
0603      MOV     R0, #0
0604      MOV     R0, #0
0605      MOV     R0, #0
0606      MOV     R0, #0
0607      MOV     R0, #0
0608      MOV     R0, #0
0609      MOV     R0, #0
0610      MOV     R0, #0
0611      MOV     R0, #0
0612      MOV     R0, #0
0613      MOV     R0, #0
0614      MOV     R0, #0
0615      MOV     R0, #0
0616      MOV     R0, #0
0617      MOV     R0, #0
0618      MOV     R0, #0
0619      MOV     R0, #0
0620      MOV     R0, #0
0621      MOV     R0, #0
0622      MOV     R0, #0
0623      MOV     R0, #0
0624      MOV     R0, #0
0625      MOV     R0, #0
0626      MOV     R0, #0
0627      MOV     R0, #0
0628      MOV     R0, #0
0629      MOV     R0, #0
0630      MOV     R0, #0
0631      MOV     R0, #0
0632      MOV     R0, #0
0633      MOV     R0, #0
0634      MOV     R0, #0
0635      MOV     R0, #0
0636      MOV     R0, #0
0637      MOV     R0, #0
0638      MOV     R0, #0
0639      MOV     R0, #0
0640      MOV     R0, #0
0641      MOV     R0, #0
0642      MOV     R0, #0
0643      MOV     R0, #0
0644      MOV     R0, #0
0645      MOV     R0, #0
0646      MOV     R0, #0
0647      MOV     R0, #0
0648      MOV     R0, #0
0649      MOV     R0, #0
0650      MOV     R0, #0
0651      MOV     R0, #0
0652      MOV     R0, #0
0653      MOV     R0, #0
0654      MOV     R0, #0
0655      MOV     R0, #0
0656      MOV     R0, #0
0657      MOV     R0, #0
0658      MOV     R0, #0
0659      MOV     R0, #0
0660      MOV     R0, #0
0661      MOV     R0, #0
0662      MOV     R0, #0
0663      MOV     R0, #0
0664      MOV     R0, #0
0665      MOV     R0, #0
0666      MOV     R0, #0
0667      MOV     R0, #0
0668      MOV     R0, #0
0669      MOV     R0, #0
0670      MOV     R0, #0
0671      MOV     R0, #0
0672      MOV     R0, #0
0673      MOV     R0, #0
0674      MOV     R0, #0
0675      MOV     R0, #0
0676      MOV     R0, #0
0677      MOV     R0, #0
0678      MOV     R0, #0
0679      MOV     R0, #0
0680      MOV     R0, #0
0681      MOV     R0, #0
0682      MOV     R0, #0
0683      MOV     R0, #0
0684      MOV     R0, #0
0685      MOV     R0, #0
0686      MOV     R0, #0
0687      MOV     R0, #0
0688      MOV     R0, #0
0689      MOV     R0, #0
0690      MOV     R0, #0
0691      MOV     R0, #0
0692      MOV     R0, #0
0693      MOV     R0, #0
0694      MOV     R0, #0
0695      MOV     R0, #0
0696      MOV     R0, #0
0697      MOV     R0, #0
0698      MOV     R0, #0
0699      MOV     R0, #0
0700      MOV     R0, #0
0701      MOV     R0, #0
0702      MOV     R0, #0
0703      MOV     R0, #0
0704      MOV     R0, #0
0705      MOV     R0, #0
0706      MOV     R0, #0
0707      MOV     R0, #0
0708      MOV     R0, #0
0709      MOV     R0, #0
0710      MOV     R0, #0
0711      MOV     R0, #0
0712      MOV     R0, #0
0713      MOV     R0, #0
0714      MOV     R0, #0
0715      MOV     R0, #0
0716      MOV     R0, #0
0717      MOV     R0, #0
0718      MOV     R0, #0
0719      MOV     R0, #0
0720      MOV     R0, #0
0721      MOV     R0, #0
0722      MOV     R0, #0
0723      MOV     R0, #0
0724      MOV     R0, #0
0725      MOV     R0, #0
0726      MOV     R0, #0
0727      MOV     R0, #0
0728      MOV     R0, #0
0729      MOV     R0, #0
0730      MOV     R0, #0
0731      MOV     R0, #0
0732      MOV     R0, #0
0733      MOV     R0, #0
0734      MOV     R0, #0
0735      MOV     R0, #0
0736      MOV     R0, #0
0737      MOV     R0, #0
0738      MOV     R0, #0
0739      MOV     R0, #0
0740      MOV     R0, #0
0741      MOV     R0, #0
0742      MOV     R0, #0
0743      MOV     R0, #0
0744      MOV     R0, #0
0745      MOV     R0, #0
0746      MOV     R0, #0
0747      MOV     R0, #0
0748      MOV     R0, #0
0749      MOV     R0, #0
0750      MOV     R0, #0
0751      MOV     R0, #0
0752      MOV     R0, #0
0753      MOV     R0, #0
0754      MOV     R0, #0
0755      MOV     R0, #0
0756      MOV     R0, #0
0757      MOV     R0, #0
0758      MOV     R0, #0
0759      MOV     R0, #0
0760      MOV     R0, #0
0761      MOV     R0, #0
0762      MOV     R0, #0
0763      MOV     R0, #0
0764      MOV     R0, #0
0765      MOV     R0, #0
0766      MOV     R0, #0
0767      MOV     R0, #0
0768      MOV     R0, #0
0769      MOV     R0, #0
0770      MOV     R0, #0
0771      MOV     R0, #0
0772      MOV     R0, #0
0773      MOV     R0, #0
0774      MOV     R0, #0
0775      MOV     R0, #0
0776      MOV     R0, #0
0777      MOV     R0, #0
0778      MOV     R0, #0
0779      MOV     R0, #0
0780      MOV     R0, #0
0781      MOV     R0, #0
0782      MOV     R0, #0
0783      MOV     R0, #0
0784      MOV     R0, #0
0785      MOV     R0, #0
0786      MOV     R0, #0
0787      MOV     R0, #0
0788      MOV     R0, #0
0789      MOV     R0, #0
0790      MOV     R0, #0
0791      MOV     R0, #0
0792      MOV     R0, #0
0793      MOV     R0, #0
0794      MOV     R0, #0
0795      MOV     R0, #0
0796      MOV     R0, #0
0797      MOV     R0, #0
0798      MOV     R0, #0
0799      MOV     R0, #0
0800      MOV     R0, #0
0801      MOV     R0, #0
0802      MOV     R0, #0
0803      MOV     R0, #0
0804      MOV     R0, #0
0805      MOV     R0, #0
0806      MOV     R0, #0
0807      MOV     R0, #0
0808      MOV     R0, #0
0809      MOV     R0, #0
0810      MOV     R0, #0
0811      MOV     R0, #0
0812      MOV     R0, #0
0813      MOV     R0, #0
0814      MOV     R0, #0
0815      MOV     R0, #0
0816      MOV     R0, #0
0817      MOV     R0, #0
0818      MOV     R0, #0
0819      MOV     R0, #0
0820      MOV     R0, #0
0821      MOV     R0, #0
0822      MOV     R0, #0
0823      MOV     R0, #0
0824      MOV     R0, #0
0825      MOV     R0, #0
0826      MOV     R0, #0
0827      MOV     R0, #0
0828      MOV     R0, #0
0829      MOV     R0, #0
0830      MOV     R0, #0
0831      MOV     R0, #0
0832      MOV     R0, #0
0833      MOV     R0, #0
0834      MOV     R0, #0
0835      MOV     R0, #0
0836      MOV     R0, #0
0837      MOV     R0, #0
0838      MOV     R0, #0
0839      MOV     R0, #0
0840      MOV     R0, #0
0841      MOV     R0, #0
0842      MOV     R0, #0
0843      MOV     R0, #0
0844      MOV     R0, #0
0845      MOV     R0, #0
0846      MOV     R0, #0
0847      MOV     R0, #0
0848      MOV     R0, #0
0849      MOV     R0, #0
0850      MOV     R0, #0
0851      MOV     R0, #0
0852      MOV     R0, #0
0853      MOV     R0, #0
0854      MOV     R0, #0
0855      MOV     R0, #0
0856      MOV     R0, #0
0857      MOV     R0, #0
0858      MOV     R0,
```

Using the ADC and PWM of the 87C752/87C752 AN428

DEMO752C-2 87C752 A/D and PWM Demonstration 12/03/90 PAGE 1

```

1  ; *****
2  ;
3  ; *****
4  ;
5  ; 87C752 A/D and PWM Demonstration Program
6  ;
7  ; This program first reads all five A/D channels and outputs the values in
8  ; hexadecimal as RS-232 data. Next, the PWM output is set to reflect the
9  ; value on A/D channel 0, and again outputs the A/D value to RS-232. Note
10 ; that the A/D value is inverted before being moved to the PWM compare
11 ; register in order to compensate for the inversion on the PWM output pin.
12 ; This process is repeated continuously.
13 ;
14 ; Thus, a voltage may be applied to ADC0 (P1.0, pin 13) to vary the PWM pulse
15 ; width. A simple test of this function is to measure the voltage on ADC0
16 ; and PWM with a voltmeter. A typical voltmeter will integrate the waveform
17 ; on the PWM output and show a voltage within about 20mV of that on ADC0.
18 ;
19 ; The RS-232 output appears on Port 1 pin 5, which must be buffered with an
20 ; MC1488 or perhaps a MAX232 chip prior to being connected to a terminal.
21 ; The transmission rate will be 9600 baud when the 87C752 is operated from
22 ; 16MHz crystal.
23 ; *****
24 ; *****
25 ;
26 $Title(87C752 A/D and PWM Demonstration)
27 $Date(12/03/90)
28 $MOD752
29 ; *****
30 ;
31
32 BaudVal EQU -139 ;Timer value for 9600 baud @ 16 MHz.
33 ;(one bit cell time)
34
35 XmtDat DATA 10h ;Data for RS-232 transmit routine.
36 BitCnt DATA 12h ;RS-232 transmit bit count.
37 PWMVal DATA 13h ;Holds next value for updating the PWM.
38 ADVal DATA 14h ;Holds last A/D conversion result.
39
40 Flags DATA 20h
41 TxFlag BIT Flags.0 ;Transmit-in-progress flag.
42 ADFlag BIT Flags.1 ;Indicates A/D conversion complete.
43
44 TxD BIT P1.5 ;Port bit for RS-232 transmit.
45
46 ; *****
47 ;
48 ; Interrupt Vectors
49
50 ORG 0 ;Reset vector.
51 AJMP Reset
52
53 ORG 0BH ;Timer 0 interrupt.
54 AJMP Timr0 ;(used as a baud rate generator)
55
56 ORG 2Bh ;A/D conversion complete interrupt.
57 AJMP ADInt
58

```


Using the ADC and PWM of the 83C752/87C752 AN428

DEMO752C 00000000

87C752 A/D and PWM Demonstration

12/03/90 PAGE 2

```

0033 59      ORG      33h      ;PWM interrupt.
0033 01A3    60      AJMP     PWMInt
                                61
                                62
                                63      ;*****
                                64
0035 758130  65      Reset:  MOV     SP,#30h
0038 752000  66      MOV     Flags,#0      ;Clear RS-232 flags.
003B 758800  67      MOV     TCON,#00h    ;Set up timer controls.
003E 75A882  68      MOV     IE,#82h    ;Enable timer 0 interrupt.
                                69
0041 90011B  70      MOV     DPTR,#Msg1      ;Point to message string.
0044 310A    71      ACALL    Mess        ;Send message.
                                72
0046 7900    73      MOV     R1,#0      ;Start with A/D channel 0.
0048 E9      74      Loop1:  MOV     A,R1
0049 118D    75      ACALL    ADConv      ;Start A/D conversion.
004B FA      76      MOV     R2,A
                                77
004C 900152  78      MOV     DPTR,#Msg2      ;Point to message string.
004F 310A    79      ACALL    Mess        ;Send message.
0051 E9      80      MOV     A,R1
0052 11EC    81      ACALL    PrByte      ;Print channel #.
0054 900161  82      MOV     DPTR,#Msg3      ;Point to message string.
0057 310A    83      ACALL    Mess        ;Send message.
                                84
0059 EA      85      MOV     A,R2
005A 11EC    86      ACALL    PrByte      ;Print A/D value.
005C 09      87      INC     R1        ;Advance R1 value.
005D B905E8  88      CJNE    R1,#5,Loop1
0060 90014F  89      MOV     DPTR,#CRLF      ;Point to message string.
0063 310A    90      ACALL    Mess        ;Send message.
                                91
                                92      ; Now use A/D channel 0 value to control the PWM.
                                93
0065 758FFF  94      MOV     PWMP,#0FFh      ;Set PWM slow frequency.
0068 758E00  95      MOV     PWCM,#0      ;Set initial PWM value.
006B 751300  96      MOV     PWMVal,#0      ;Default starting value for the PWM.
006E 75FE01  97      MOV     PWENA,#1      ;Start PWM
0071 75A8CA  98      MOV     IE,#0CAh      ;Now enable the A/D and PWM interrupts.
                                99
0074 7400    100     Loop2:  MOV     A,#0      ;Read A/D channel 0.
0076 1186    101     ACALL    ADStart      ;Start A/D conversion.
0078 3001FD  102     JNB     ADFlag,$      ;Wait for A/D conversion complete.
007B E514    103     MOV     A,ADVal      ;Get A/D result to print.
007D 11EC    104     ACALL    PrByte      ;Print PWM value.
007F 900165  105     MOV     DPTR,#Msg4      ;Point to message string.
0082 310A    106     ACALL    Mess        ;Send message.
0084 80EE    107     SJMP     Loop2
                                108
                                109     ; A/D Conversion Routines.
                                110
                                111     ; The following shows two ways to use the A/D. Both routines are used by
                                112     ; different portions of the sample program.
                                113
                                114     ; Method 1: This version of the routine starts the conversion and then
                                115     ; returns. The mainline program can detect when the conversion is
                                116     ; complete by checking the A/D conversion complete flag (ADFlag) which is

```

```

117 ; set by the A/D interrupt service routine. A/D data must be read by the
118 ; calling routine.
119
0086 C201 120 ADStart: CLR ADFlag ;Clear A/D conversion complete flag.
0088 4428 121 ORL A,#28h ;Add control bits to channel #.
008A F5A0 122 MOV ADCON,A ;Start conversion.
008C 22 123 RET
124 ; Clear RS-232 flag
125 ; Add up timer count
126 ; Method 2: This is an alternative version of the A/D routine which
127 ; starts the conversion and then waits for it to complete before
128 ; returning. A/D data is returned in the ACC.
129 ; Send message
130 ADConv: ORL A,#28h ;Add control bits to channel #.
008D 4428 131 MOV ADCON,A ;Start conversion.
008F F5A0 132 ADC1: MOV A,ADCON ;Wait for conversion complete.
0091 E5A0 133 JNB ACC.4,ADC1 ;Read A/D.
0093 30E4FB 134 MOV A,ADAT
0096 E584 135 RET
0098 22 136 ; A/D interrupt service routine.
137 ; Send message
138 ; A/D interrupt service routine.
139 ; Send channel #
0099 E584 140 ADInt: MOV A,ADAT ;Read A/D data.
009B F514 141 MOV ADVal,A ;Save A/D data for print routine.
009D F4 142 CPL A ;Complement the value for the PWM.
009E F513 143 MOV PWMVal,A ;Set new value for PWM update.
00A0 D201 144 SETB ADFlag ;Tell main that new A/D data is ready.
00A2 32 145 RETI
146 ; Point to message
147 ; PWM interrupt service routine allows updating the PWM synchronously.
148 ; PWM interrupt service routine allows updating the PWM synchronously.
149
00A3 85138E 150 PWMInt: MOV PWCM,PWMVal ;Update PWM duty cycle.
00A6 32 151 RETI
152 ; Set PWM value
153 ; Set initial PWM value
154 ; Send a byte out RS-232 and wait for completion before returning.
155 ; Send PWM
00A7 11AD 156 XmtByte: ACALL RSXmt ;Send ACC to RS-232 output.
00A9 2000FD 157 JB TxFlag,$ ;Wait for transmit complete.
00AC 22 158 RET
159 ; Start A/D conversion
160 ; Wait for A/D conversion to complete
161 ; Begin RS-232 transmit.
162 ; Send message
00AD F510 163 RSXmt: MOV XmtDat,A ;Save data to be transmitted.
00AF 75120A 164 MOV BitCnt,#10 ;Set bit count.
00B2 758CFF 165 MOV TH,#High BaudVal ;Set timer for baud rate.
00B5 758A75 166 MOV TL,#Low BaudVal
00B8 758DFF 167 MOV RTH,#High BaudVal ;Also set timer reload value.
00BB 758B75 168 MOV RTL,#Low BaudVal
00BE D28C 169 SETB TR ;Start timer.
00C0 C295 170 CLR TxD ;Begin start bit.
00C2 D200 171 SETB TxFlag ;Set transmit-in-progress flag.
00C4 22 172 RET
173 ; Method 1: This version of the routine for the conversion and then
174 ; returns the A/D data in the ACC.
175 ; Complete by checking the A/D conversion complete flag which is

```

Using the ADC and PWM of the 83C752/87C752 to AN428

DEMO752C

87C752 A/D and PWM Demonstration

12/03/90 PAGE 4

```

175 ; Timer 0 timeout: RS-232 receive bit or transmit bit.
176
00C5 C0E0 177 Timr0: PUSH ACC
00C7 C0D0 178 PUSH PSW
00C9 200007 179 JB TxFlag,TxBit ;Is this a transmit timer interrupt?
00CC C28C 180 T0Ex1: CLR TR ;Stop timer.
00CE D0D0 181 T0Ex2: POP PSW
00D0 D0E0 182 POP ACC
00D2 32 183 RETI
184
185
186 ; RS-232 transmit bit routine.
187
00D3 D51204 188 TxBit: DJNZ BitCnt,TxBusy ;Decrement bit count, test for done.
00D6 C200 189 CLR TxFlag ;End of stop bit, release timer.
00D8 80F2 190 SJMP T0Ex1 ;Stop timer and exit.
191
00DA E512 192 TxBusy: MOV A,BitCnt ;Get bit count.
00DC B40104 193 CJNE A,#1,TxNext ;Is this a stop bit?
00DF D295 194 SETB TxD ;Set stop bit.
00E1 80EB 195 SJMP T0Ex2 ;Exit.
196
00E3 E510 197 TxNext: MOV A,XmtDat ;Get data.
00E5 13 198 RRC A ;Advance to next bit.
00E6 F510 199 MOV XmtDat,A
00E8 9295 200 MOV TxD,C ;Send data bit.
00EA 80E2 201 SJMP T0Ex2 ;Exit.
202
203
204 ; Print byte routine: print ACC contents as ASCII hexadecimal.
205
00EC C0E0 206 PrByte: PUSH ACC
00EE C4 207 SWAP A
00EF 11FA 208 ACALL HexAsc
00F1 11A7 209 ACALL XmtByte
00F3 D0E0 210 POP ACC
00F5 11FA 211 ACALL HexAsc ;Print nibble in ACC as ASCII hex.
00F7 11A7 212 ACALL XmtByte
00F9 22 213 RET
214
215
216 ; Hexadecimal to ASCII conversion routine.
217
00FA 540F 218 HexAsc: ANL A,#0FH ;Convert a nibble to ASCII hex.
00FC 30E308 219 JNB ACC.3,NoAdj
00FF 20E203 220 JB ACC.2,Adj
0102 30E102 221 JNB ACC.1,NoAdj
0105 2407 222 Adj: ADD A,#07H
0107 2430 223 NoAdj: ADD A,#30H
0109 22 224 RET
225
226
227 ; Message string transmit routine.
228
010A C0E0 229 Mess: PUSH ACC
010C 7800 230 MOV R0,#0 ;R0 is character pointer (string
010E E8 231 Mesl: MOV A,R0 ; length is limited to 256 bytes).
010F 93 232 MOVC A,@A+DPTR ;Get byte to send.

```

Using the ADC and PWM of the 83C752/87C752 AN428

DEMO752C 09/03/90

87C752 A/D and PWM Demonstration

12/03/90 PAGE 5

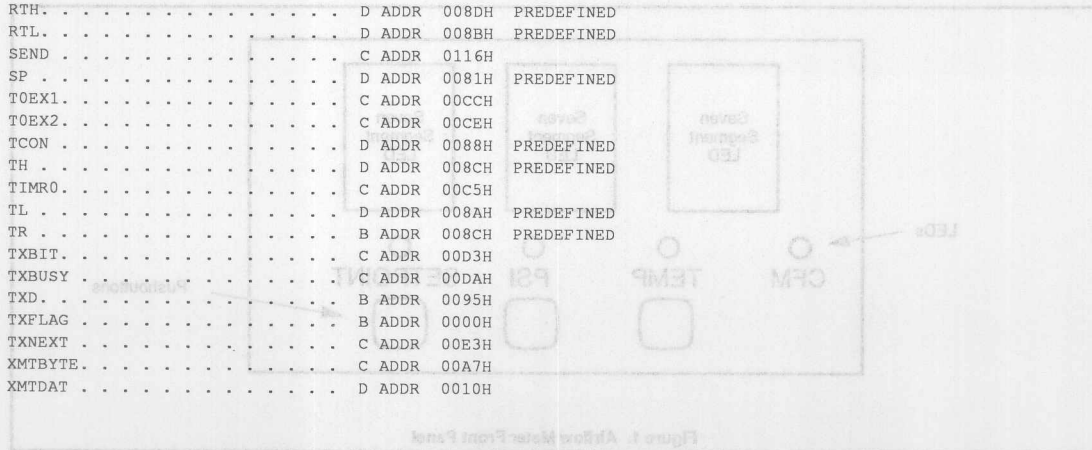
```

0110 B40003      233      CJNE      A,#0,Send      ;End of string is indicated by a 0.
0113 D0E0        234      POP      ACC
0115 22          235      RET
                  236
0116 11A7        237      Send:    ACALL   XmtByte    ;Send a character.
0118 08          238      INC      R0              ;Next character.
0119 80F3        239      SJMP     Mes1
                  240
011B 0D0A        241      Msg1:    DB      0Dh, 0Ah,
011D 54686973    242      DB      'This is a demonstration of the 87C752 A/D and PWM.'
0121 20697320
0125 61206465
0129 6D6F6E73
012D 74726174
0131 696F6E20
0135 6F662074
0139 68652038
013D 37433735
0141 3220412F
0145 4420616E
0149 64205057
014D 4D2E
014F 0D0A00      243      CRLF:    DB      0Dh, 0Ah, 0
                  244
0152 0D0A412F    245      Msg2:    DB      0Dh, 0Ah, 'A/D Channel ', 0
0156 44204368
015A 616E6E65
015E 6C2000
                  246
0161 203D2000    247      Msg3:    DB      ' = ', 0
                  248
0165 202000      249      Msg4:    DB      ' , ', 0
                  250
                  251      END

ASSEMBLY COMPLETE, 0 ERRORS FOUND

```

ACC	D ADDR 00E0H	PREDEFINED
ADAT	D ADDR 0084H	PREDEFINED
ADC1	C ADDR 0091H	
ADCON	D ADDR 00A0H	PREDEFINED
ADCONV	C ADDR 008DH	
ADFLAG	B ADDR 0001H	
ADINT	C ADDR 0099H	
ADJ	C ADDR 0105H	
ADSTART	C ADDR 0086H	
ADVAL	D ADDR 0014H	
BAUDVAL	NUMB FF75H	
BITCNT	D ADDR 0012H	
CRLF	C ADDR 014FH	
FLAGS	D ADDR 0020H	
HEXASC	C ADDR 00FAH	
IE	D ADDR 00A8H	PREDEFINED
LOOP1	C ADDR 0048H	
LOOP2	C ADDR 0074H	
MESL	C ADDR 010EH	
MESS	C ADDR 010AH	
MSG1	C ADDR 011BH	
MSG2	C ADDR 0152H	
MSG3	C ADDR 0161H	
MSG4	C ADDR 0165H	
NOADJ	C ADDR 0107H	
P1	D ADDR 0090H	PREDEFINED
PRBTYPE	C ADDR 00ECH	
PSW	D ADDR 00D0H	PREDEFINED
PWCM	D ADDR 008EH	PREDEFINED
PWENA	D ADDR 00FEH	PREDEFINED
PWMINT	C ADDR 00A3H	
PWMP	D ADDR 008FH	PREDEFINED
PWMVAL	D ADDR 0013H	
RESET	C ADDR 0035H	
RSXMT	C ADDR 00ADH	
RTH	D ADDR 008DH	PREDEFINED
RTL	D ADDR 008BH	PREDEFINED
SEND	C ADDR 0116H	
SP	D ADDR 0081H	PREDEFINED
TOEX1	C ADDR 00CCH	
TOEX2	C ADDR 00CEH	
TCON	D ADDR 0088H	PREDEFINED
TH	D ADDR 008CH	PREDEFINED
TIMR0	C ADDR 00C5H	
TL	D ADDR 008AH	PREDEFINED
TR	B ADDR 008CH	PREDEFINED
TXBIT	C ADDR 00D0H	
TXBUSY	C ADDR 00DAH	
TXD	B ADDR 0095H	
TXFLAG	B ADDR 0000H	
TXNEXT	C ADDR 00E3H	
XMTBYTE	C ADDR 00A7H	
XMTDAT	D ADDR 0010H	



Airflow measurement using the 83/87C752 and "C" AN429

INTRODUCTION

This application note describes a low-cost airflow measurement device based on the Philips 83/87C752 microcontroller. Airflow measurement—determining the volume of air transferred per unit time (cubic feet per minute, or cfm)—is intrinsic to a variety of industrial and scientific processes.

Airflow computation depends on three simultaneous physical air measurements—velocity, pressure, and temperature. This design includes circuits and sensors allowing the 8XC752 to measure all three parameters.

The design also includes seven-segment LED displays, discrete LEDs, and pushbutton switches to allow selective display of airflow, temperature, and pressure. Furthermore, airflow is continuously compared with a programmer-defined setpoint. Should the measured airflow exceed the setpoint, an output relay is energized. In actual application, this relay output could be used to signal the setpoint violation (via lamp or audio annunciator) or otherwise control the overall process (e.g., emergency process shutdown). Of course, the setpoint, comparison criteria (greater, less than, etc.) and violation response (relay on, relay off) are easily changed by program modification to meet actual application requirements.

Referring to Figure 1, the overall operation of the airflow device is as follows.

Normally the unit continuously displays the airflow (in cfm) on the seven-segment

displays. The discrete CFM LED is also lit to confirm the parameter being displayed.

Pressing the TEMP pushbutton switches the display to temperature (in degrees C) and lights the TEMP LED. As long as the pushbutton remains pressed, the temperature is displayed. When the pushbutton is released, the display reverts to the default pressure display.

Similarly, pressing the PSI pushbutton displays the atmospheric pressure (in pounds per square inch) and lights the PSI LED. The pressure is displayed as long as the pushbutton is pressed, and the default airflow display resumes when the pushbutton is released.

Finally, pressing the SET-POINT pushbutton displays the programmed airflow setpoint (in cfm) and lights the SET-POINT LED. Again, releasing the pushbutton causes the display to revert to the default airflow measurement.

CONTROL PROGRAMMING IN "C"

While, thanks to advanced semiconductor processing, hardware price/performance continues to improve, software development technology has changed little over time. Thus, given ever-rising costs for qualified personnel, software "productivity" is arguably in decline. Indeed, for low-unit cost and/or low-volume applications, software development has emerged as the major portion of total design cost. Furthermore,

beyond the initial programming cost, "hidden" costs also arise in the form of life-cycle code maintenance and revision and lost revenue/market share due to excessive time-to-market.

Traditionally, control applications have been programmed in assembly language to overcome microcontroller resource and performance constraints. Now, thanks to more powerful microcontrollers and advanced compiler technology, it is feasible to program control applications using a High-Level Language (HLL).

The primary benefit of using an HLL is obvious—one HLL program "statement" can perform the same function as many lines of assembly language. Furthermore, a well-written HLL program will typically be more "readable" than an assembly language equivalent, resulting in reduced maintenance and revision/upgrade costs.

Of the many popular HLLs, the "C" language has emerged as the major contender for control applications. More than other languages, C gives the programmer direct access to, and control of, low-level hardware resources—a requirement for deterministic, real-time I/O applications. Furthermore, C is based on a "minimalist" philosophy in which the language performs only those functions explicitly requested by the programmer. This approach is well-suited for control applications, which are often characterized by strict cost and performance requirements.

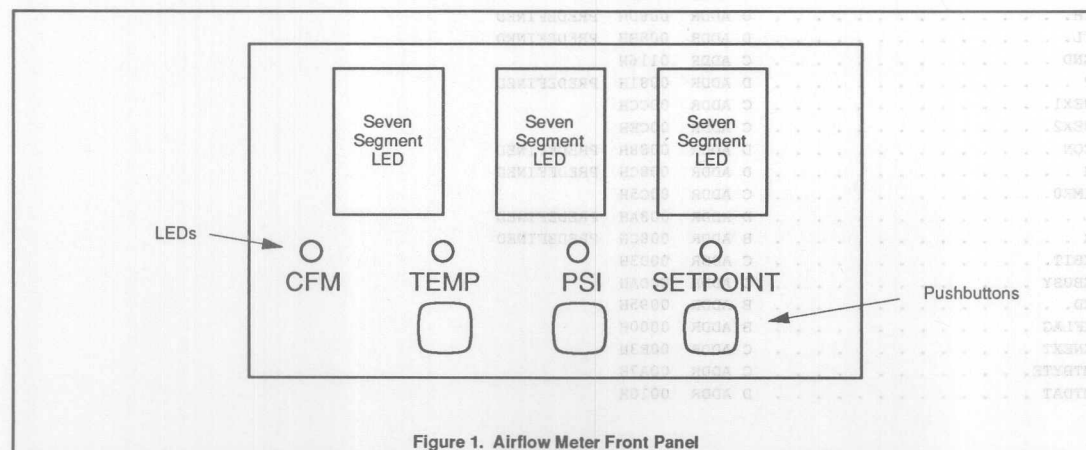


Figure 1. Airflow Meter Front Panel

Airflow measurement using the 83/87C752 and "C" AN429

8XC752 OVERVIEW

The 83C752/87C752 (ROM/EPROM-based) combine the performance advantages of the 8-bit 80C51 architecture with the low cost, power consumption, and size/pin count of a 4-bit microcontroller. Therefore, the 8XC752 is uniquely capable of bringing high processing speed and HLL programming to even the most cost-sensitive applications such as handheld (battery driven) instruments, automotive distributed processing, "smart" appliances, and sophisticated consumer electronics.

Obviously, the 8XC752 can be used for cost-reduced versions of existing 8-bit applications. The device can also replace similarly priced 4-bit devices to achieve benefits of higher performance and, most importantly, easier s/w development including the use of HLL. Indeed, the component and system design costs associated with the 8XC752 are so low that it is a viable candidate for first-time computerization of formerly non-microcontroller-based designs.

Figure 2 shows the block diagram of the 8XC752. Major features of the device include the following.

Full-Function, High-Speed (to 16MHz) 80C51 CPU Core

The popular 80C51 architecture features 8- and 16-bit processing and high-speed execution. Most instructions execute in a single machine cycle (the slowest instructions require only two cycles). Though a

streamlined architecture, the CPU core, unlike 4-bit devices, includes all the basic capabilities (such as stack, multiply instruction, interrupts, etc.) required to support HLL compilation. The CPU core also includes a unique Boolean processor which is well-suited for the bit-level processing and I/O common to control applications.

Low-Power CMOS and Power-Saving Operation Modes

Thanks to the advanced CMOS process, the 8XC752 features extremely low power consumption, which helps to extend battery life in handheld applications and otherwise reduce power supply and thermal dissipation costs and reliability concerns. Low ACTIVE mode (full-speed operation) power consumption—only 11mA typical at 12MHz—is further complemented by two program-initiated power-saving operation modes—IDLE and POWER-DOWN.

In idle mode, CPU instruction processing stops while on-chip I/O and RAM remain powered. Power consumption drops to 1.5µA (typical, 12MHz) until processing is restarted by interrupt or reset. Power-down mode cuts power consumption further (to only 10µA typical at 12MHz) by stopping both instruction and I/O processing. Return to full-speed operation from power-down mode is via reset.

Note that power consumption can be further cut by reducing the clock frequency as much as application performance requirements allow, as shown in Figure 3.

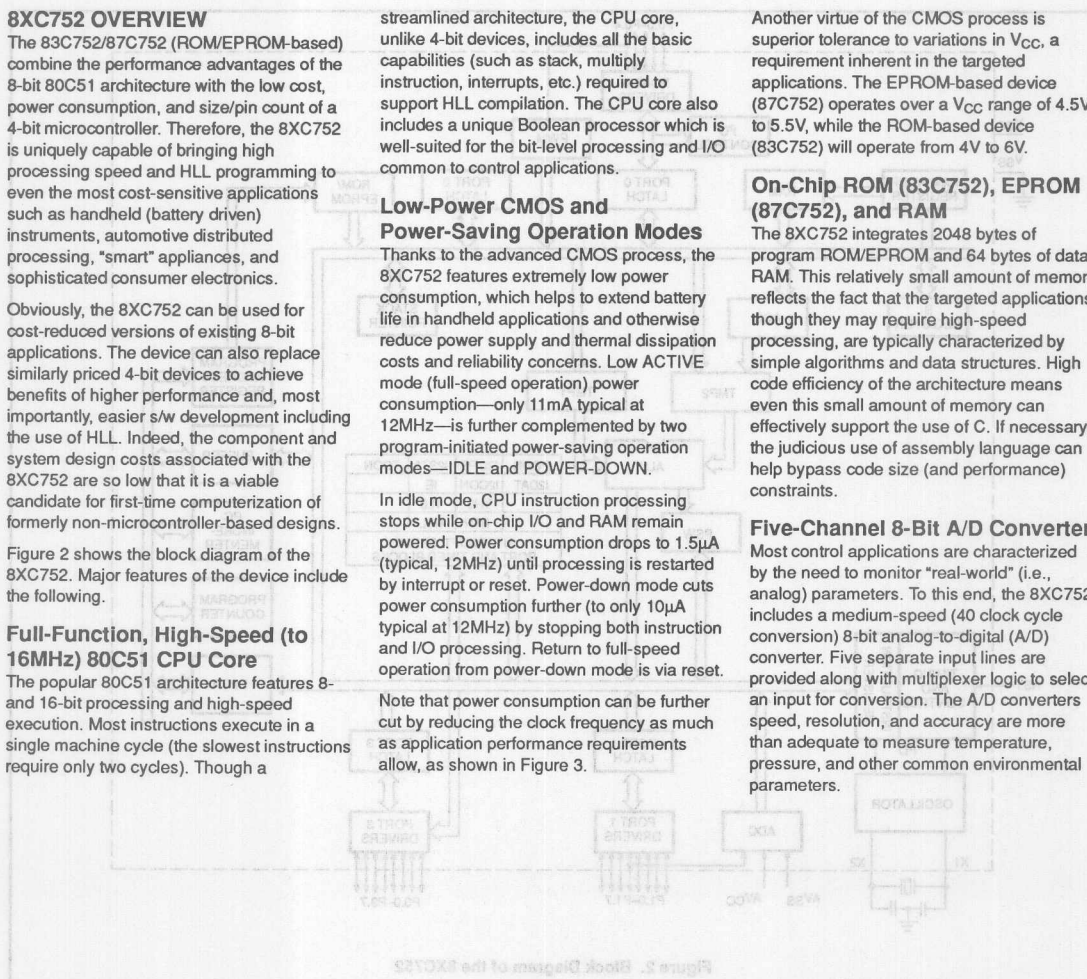
Another virtue of the CMOS process is superior tolerance to variations in V_{CC} , a requirement inherent in the targeted applications. The EPROM-based device (87C752) operates over a V_{CC} range of 4.5V to 5.5V, while the ROM-based device (83C752) will operate from 4V to 6V.

On-Chip ROM (83C752), EPROM (87C752), and RAM

The 8XC752 integrates 2048 bytes of program ROM/EPROM and 64 bytes of data RAM. This relatively small amount of memory reflects the fact that the targeted applications, though they may require high-speed processing, are typically characterized by simple algorithms and data structures. High code efficiency of the architecture means even this small amount of memory can effectively support the use of C. If necessary, the judicious use of assembly language can help bypass code size (and performance) constraints.

Five-Channel 8-Bit A/D Converter

Most control applications are characterized by the need to monitor "real-world" (i.e., analog) parameters. To this end, the 8XC752 includes a medium-speed (40 clock cycle conversion) 8-bit analog-to-digital (A/D) converter. Five separate input lines are provided along with multiplexer logic to select an input for conversion. The A/D converters speed, resolution, and accuracy are more than adequate to measure temperature, pressure, and other common environmental parameters.



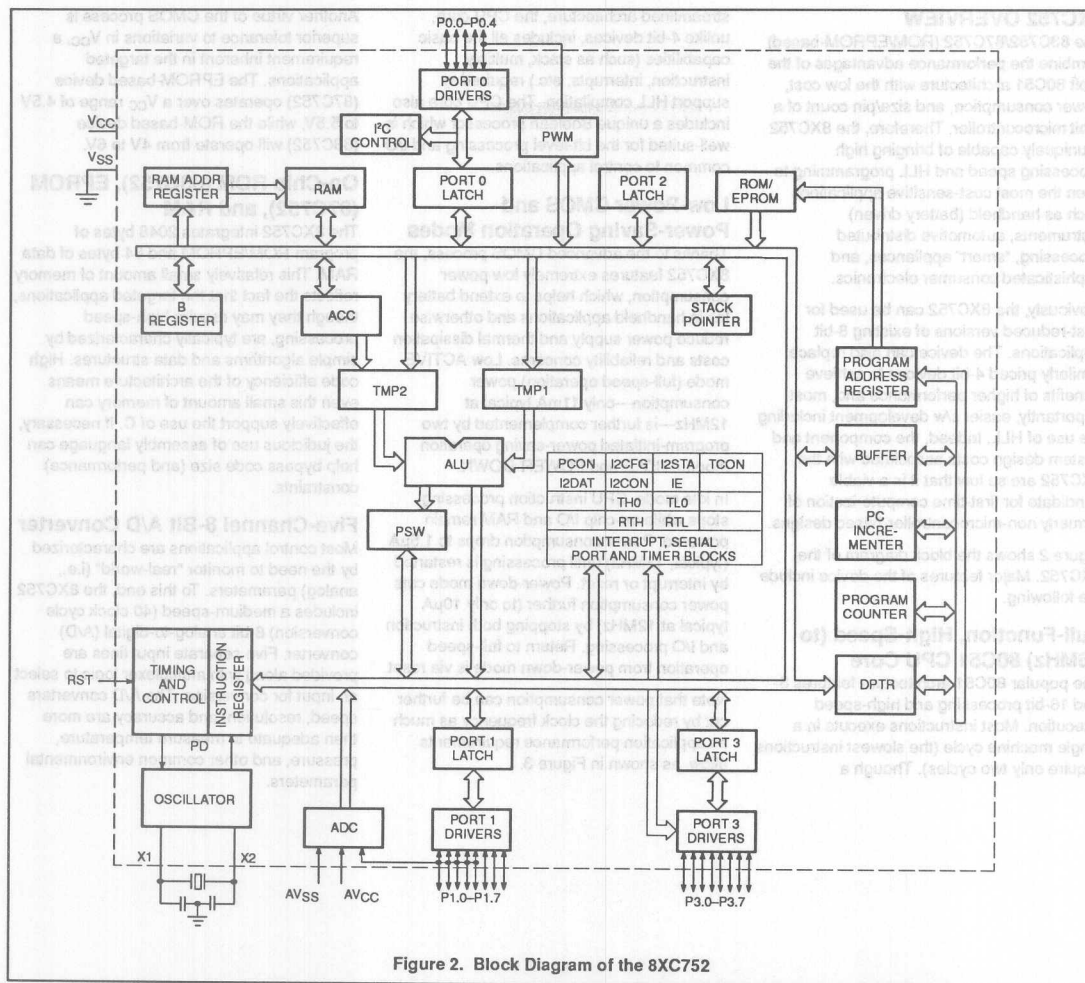


Figure 2. Block Diagram of the 8XC752

Timer/Counters

Control applications, due to their "real-time" nature, invariably call for a variety of timing and counting capabilities. The 8XC752 meets the need by integrating three separate functions—a 16-bit auto-reload counter/timer, an 8-bit pulse width modulator (PWM) output/timer, and a fixed-rate timer for timebase generation. Together, these timing/counting resources can serve a range of tasks, including waveform generation, external event counting, elapsed time

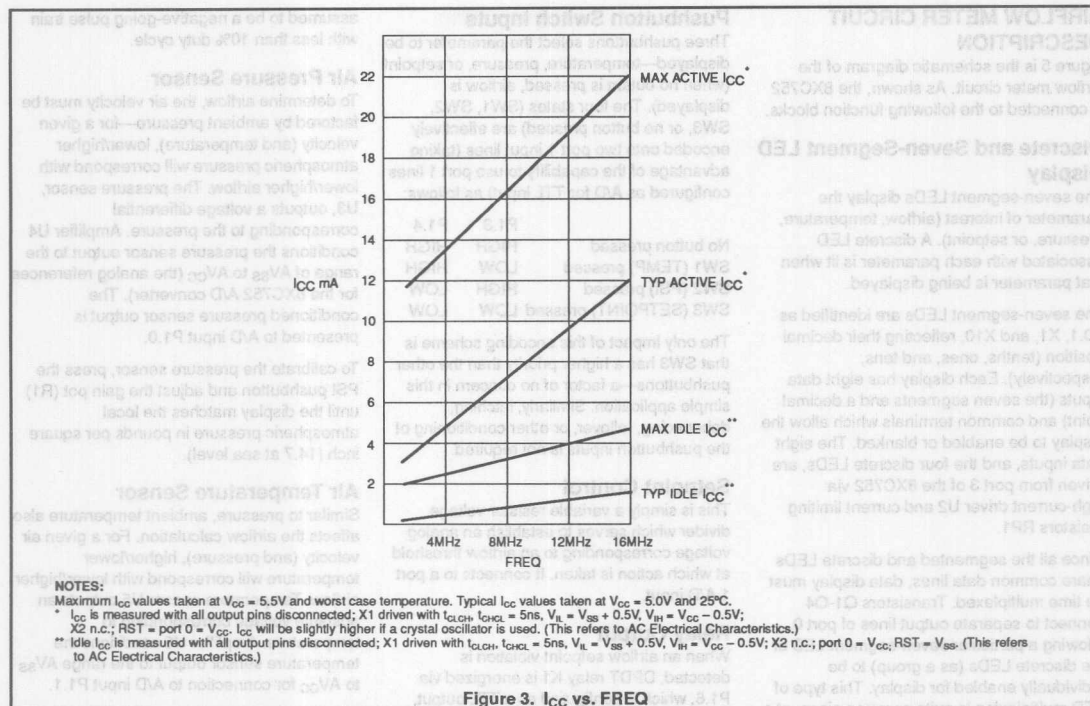
calculation, periodic interrupt generation, and watchdog timer.

I²C Bus

The Inter-Integrated Circuit (I²C) bus is a patented serial peripheral interface. The virtue of I²C is the ability to expand system functionality with acceptable performance and minimum cost. Notably, the pin and interconnect count is radically reduced compared to expansion via a typical microprocessor bus—I²C requires only two

lines, while a parallel bus often consumes 20-30 lines and may call for extra glue logic (decoder, address latch, etc.). The 8XC752 I²C port allows easy connection to a wide variety of compatible peripherals such as LCD drivers, A/D and D/A converters, consumer/telecom and special-purpose memory (e.g., EEPROM). I²C can also be used to build distributed processing systems connecting multiple I²C-compatible microcontrollers.

Airflow measurement using the 83/87C752 and "C" AN429

Figure 3. I_{CC} vs. FREQ

8XC752 PIN FUNCTIONS

Since the 8XC752 is packaged in a cost/space-saving 28-pin package DIP or PLCC), a flexible mapping of I/O functions to pins is required to ensure the widest possible application coverage.

Of the 28 pins, seven pins are allocated to basic functions, including digital power (V_{CC} , V_{SS}), analog reference (AV_{CC} , AV_{SS}), clock oscillator (X1, X2), and reset (RST). Thus, 21 pins, organized into three ports (5-bit port 0, 8-bit ports 1 and 3), are available for user I/O.

Figure 4 shows the alternative uses for these 21 lines. As shown, the mapping is quite versatile, which maximizes the access to on-chip I/O functions and helps ensure full pin utilization.

P0.0	TTL IN/OUT (open drain), I^2C clock (SCLK)
P0.1	TTL IN/OUT (open drain), I^2C data (SDA)
P0.2	TTL IN/OUT (open drain)
P0.3	TTL IN/OUT (internal pull-up)
P0.4	TTL IN/OUT (internal pull-up), PWM output
P1.0	TTL IN/OUT (internal pull-up), A/D input channel 0
P1.1	TTL IN/OUT (internal pull-up), A/D input channel 1
P1.2	TTL IN/OUT (internal pull-up), A/D input channel 2
P1.3	TTL IN/OUT (internal pull-up), A/D input channel 3
P1.4	TTL IN/OUT (internal pull-up), A/D input channel 4
P1.5	TTL IN/OUT (internal pull-up), INTO interrupt input
P1.6	TTL IN/OUT (internal pull-up), INT1 interrupt input
P1.7	TTL IN/OUT (internal pull-up), TIMER 0 (T0) input
NOTE: P1.0-P1.4 may only be changed as a group, i.e., either all TTL I/O or all A/D inputs. However, when selected as A/D inputs, P1.0-P1.4 may also be used as TTL inputs.	
P3.0	TTL IN/OUT (internal pull-up)
P3.1	TTL IN/OUT (internal pull-up)
P3.2	TTL IN/OUT (internal pull-up)
P3.3	TTL IN/OUT (internal pull-up)
P3.4	TTL IN/OUT (internal pull-up)
P3.5	TTL IN/OUT (internal pull-up)
P3.6	TTL IN/OUT (internal pull-up)
P3.7	TTL IN/OUT (internal pull-up)

Figure 4. 8XC752 I/O Port Description

Airflow measurement using the 83/87C752 and "C" AN429

AIRFLOW METER CIRCUIT DESCRIPTION

Figure 5 is the schematic diagram of the airflow meter circuit. As shown, the 8XC752 is connected to the following function blocks.

Discrete and Seven-Segment LED Display

The seven-segment LEDs display the parameter of interest (airflow, temperature, pressure, or setpoint). A discrete LED associated with each parameter is lit when that parameter is being displayed.

The seven-segment LEDs are identified as X0.1, X1, and X10, reflecting their decimal position (tenths, ones, and tens, respectively). Each display has eight data inputs (the seven segments and a decimal point) and common terminals which allow the display to be enabled or blanked. The eight data inputs, and the four discrete LEDs, are driven from port 3 of the 8XC752 via high-current driver U2 and current limiting resistors RP1.

Since all the segmented and discrete LEDs share common data lines, data display must be time multiplexed. Transistors Q1-Q4 connect to separate output lines of port 0, allowing a particular seven-segment LED or the discrete LEDs (as a group) to be individually enabled for display. This type of LED multiplexing is quite common since, at a fast enough refresh rate, the switching between displays is not perceptible by the operator. The major benefit is the reduction of I/O lines required (without multiplexing, 28, rather than 8, data lines would be required).

Pushbutton Switch Inputs

Three pushbuttons select the parameter to be displayed—temperature, pressure, or setpoint (when no button is pressed, airflow is displayed). The four states (SW1, SW2, SW3, or no button pressed) are effectively encoded onto two port 1 input lines (taking advantage of the capability to use port 1 lines configured as A/D for TTL input) as follows:

	P1.3	P1.4
No button pressed	HIGH	HIGH
SW1 (TEMP) pressed	LOW	HIGH
SW2 (PSI) pressed	HIGH	LOW
SW3 (SETPOINT) pressed	LOW	LOW

The only impact of this encoding scheme is that SW3 has a higher priority than the other pushbuttons—a factor of no concern in this simple application. Similarly, latching, debouncing, rollover, or other conditioning of the pushbutton inputs is not required.

Setpoint Control

This is simply a variable resistor voltage divider which serves to establish an analog voltage corresponding to an airflow threshold at which action is taken. It connects to a port 1 A/D input.

Relay Output

When an airflow setpoint violation is detected, DPDT relay K1 is energized via P1.6, which is configured as a TTL output, buffered by transistor Q5.

Flowmeter Input

Measurement of the air velocity is via an air turbine tachometer connected, via optoisolator U7, to P1.5, which is configured as a TTL input. The tachometer input is

assumed to be a negative-going pulse train with less than 10% duty cycle.

Air Pressure Sensor

To determine airflow, the air velocity must be factored by ambient pressure—for a given velocity (and temperature), lower/higher atmospheric pressure will correspond with lower/higher airflow. The pressure sensor, U3, outputs a voltage differential corresponding to the pressure. Amplifier U4 conditions the pressure sensor output to the range of AV_{SS} to AV_{CC} (the analog references for the 8XC752 A/D converter). The conditioned pressure sensor output is presented to A/D input P1.0.

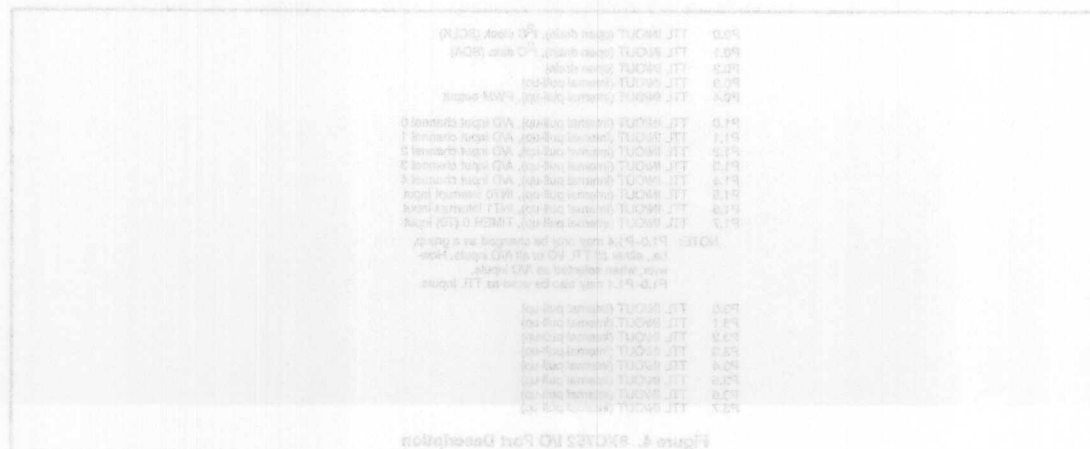
To calibrate the pressure sensor, press the PSI pushbutton and adjust the gain pot (R1) until the display matches the local atmospheric pressure in pounds per square inch (14.7 at sea level).

Air Temperature Sensor

Similar to pressure, ambient temperature also affects the airflow calculation. For a given air velocity (and pressure), higher/lower temperature will correspond with lower/higher airflow. Temperature sensor U5 outputs an absolute voltage corresponding to temperature. Amplifier U6 conditions the temperature sensor output to the range AV_{SS} to AV_{CC} for connection to A/D input P1.1.

To calibrate the temperature sensor, adjust the gain pot (R5) so that the display (while pressing the TEMP pushbutton) matches the measured output of U5 (LM35).

Figure 6 summarizes the usage of the 8XC752 I/O lines in this application.



Airflow measurement using the 83/87C752 and "C" AN429

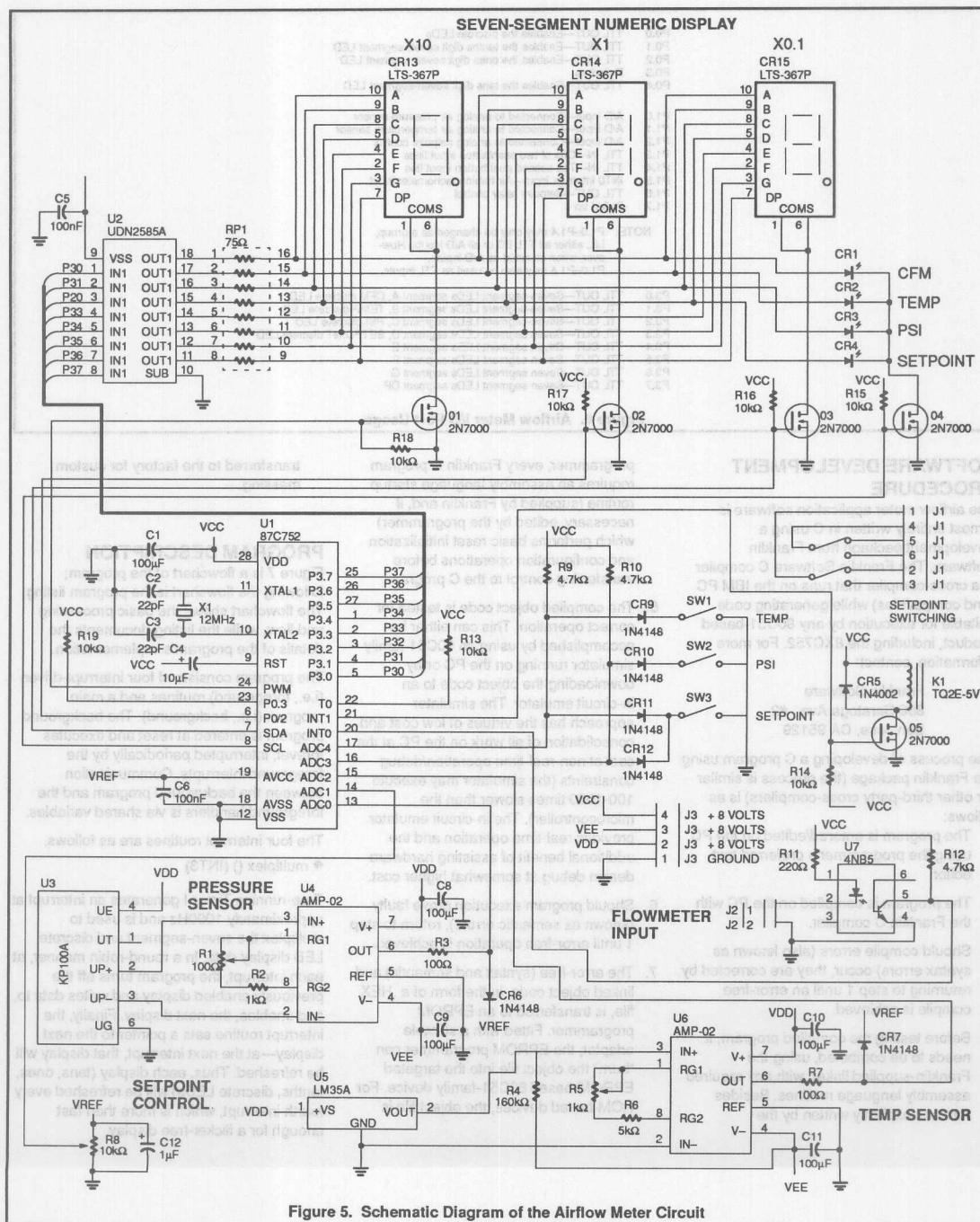


Figure 5. Schematic Diagram of the Airflow Meter Circuit

Airflow measurement using the 83/87C752 and "C" AN429

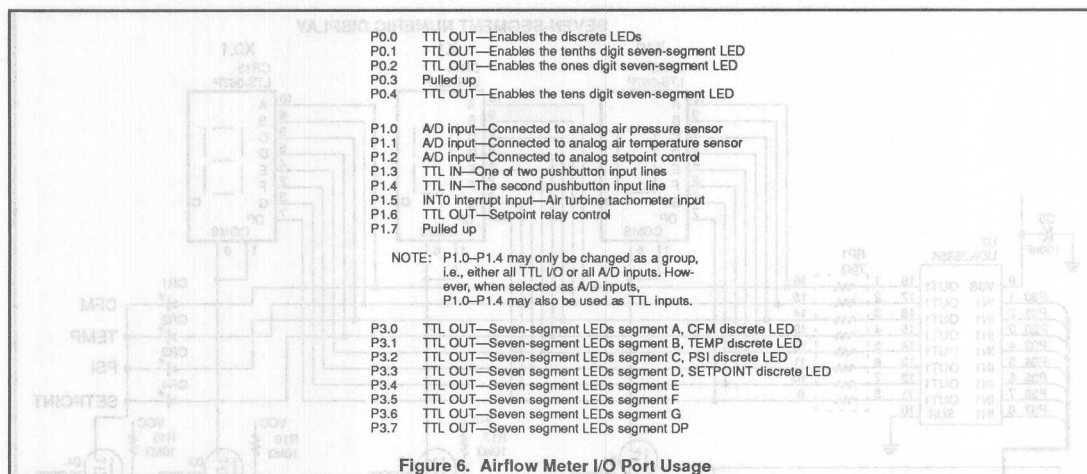


Figure 6. Airflow Meter I/O Port Usage

SOFTWARE DEVELOPMENT PROCEDURE

The airflow meter application software is almost entirely written in C using a development package from Franklin Software. The Franklin Software C compiler is a cross-compiler that runs on the IBM PC (and compatibles) while generating code suitable for execution by any 80C51-based product, including the 8XC752. For more information, contact:

Franklin Software
888 Saratoga Ave., #2
San Jose, CA 95129

The process of developing a C program using the Franklin package (the process is similar for other third-party cross-compilers) is as follows:

1. The program is entered/edited on the PC using the programmer's preferred text editor.
2. The program is compiled on the PC with the Franklin C compiler.
3. Should compile errors (also known as syntax errors) occur, they are corrected by returning to step 1 until an error-free compile is achieved.
4. Before testing the compiled program, it needs to be combined, using the Franklin-supplied linker, with any required assembly language routines. Besides routines explicitly written by the programmer, every Franklin C program requires an assembly language startup routine (supplied by Franklin and, if necessary, edited by the programmer) which performs basic reset initialization and configuration operations before transferring control to the C program.
5. The compiled object code is tested for correct operation. This can either be accomplished by using an 80C51-family simulator running on the PC or by downloading the object code to an in-circuit emulator. The simulator approach has the virtues of low cost and consolidation of all work on the PC at the cost of non-real-time operation/debug constraints (the simulator may execute 100–1000 times slower than the microcontroller). The in-circuit emulator provides real-time operation and the additional benefit of assisting hardware design debug at somewhat higher cost.
6. Should program execution prove faulty (known as semantic errors), return to step 1 until error-free operation is achieved.
7. The error-free (syntax and semantic) and linked object code, in the form of a .HEX file, is transferred to an EPROM programmer. Fitted with a suitable adaptor, the EPROM programmer can "burn" the object file into the targeted EPROM-based 80C51-family device. For ROM-based devices, the object file is

transferred to the factory for custom masking.

PROGRAM DESCRIPTION

Figure 7 is a flowchart of the program; following the flowchart is the program listing. The flowchart shows the basic processing and flow, while the listing documents the details of the program's implementation.

The program consists of four interrupt-driven (i.e., foreground) routines and a main program (i.e., background). The background program is entered at reset and executes forever, interrupted periodically by the foreground interrupts. Communication between the background program and the foreground handlers is via shared variables.

The four interrupt routines are as follows.

- multiplex () (INT3)

Free-running Timer 1 generates an interrupt at approximately 1000Hz and is used to multiplex the seven-segment and discrete LED display data. In a round-robin manner, at each interrupt, the program turns off the previously enabled display and writes data to, and enables, the next display. Finally, the interrupt routine sets a pointer to the next display—at the next interrupt, that display will be refreshed. Thus, each display (tens, ones, tenths, discrete LEDs) will be refreshed every fourth interrupt, which is more than fast enough for a flicker-free display.

Airflow measurement using the 83/87C752 and "C" AN429

• read_switch () (INT6)

The PWM prescaler is configured to generate a periodic interrupt (INT6) at about 97Hz. The program counts these interrupts, and every 32nd interrupt sets an "update" variable. The main program will change the display data when it detects that "update" is set and clear "update" to prepare for the next display cycle. Thus, display change frequency is about 33Hz (i.e., 33ms), which eliminates display glitches associated with pushbutton switch bounce.

• calc_cfm () (INT0)

The air velocity turbine tachometer drives the 8XC752 INT0 interrupt pin. At each interrupt, the program reads Timer 0, which keeps track of the elapsed time (the low 16 bits of a 24-bit count in microseconds) between INT0 interrupts. The high-order 8-bit elapsed time

count is cleared for possible updating by the following routine.

• overflow () (INT1)

When Timer 0 overflows (generating an interrupt), the program increments the high-order 8 bits of a 24-bit variable, counting the microseconds between tachometer interrupts (handled by the previous routine). If this 8-bit value becomes too large (i.e., tachometer interrupts stop), a NOFLOW variable is set, which will cause the main program to display an EEE out-of-range indicator on the seven-segment LEDs.

With the interrupt handlers executing the low-level timing and I/O, the main program, which is entered on reset and executes forever, consists of only three major steps.

The temperature/pressure compensated airflow is calculated. First, the "base" cfm rate, as tracked by the calc_cfm ()

tachometer interrupt is adjusted by removing the execution time of the calc_cfm () handler itself. Next, the temperature is determined (A/D channel 1), and airflow is compensated. Similarly, the air pressure is determined (A/D channel 0) and airflow compensated again.

Now that the true airflow is calculated, it is compared with the setpoint (adjusted with the variable resistor), which is determined by reading A/D channel 2. If the airflow is greater than the setpoint, the relay is closed. Otherwise, the relay is opened.

Finally, the UPDATE flag (set by the 33Hz read_switch () interrupt) is checked. If it is time to update, the data to be displayed is determined based on the pushbutton status and the state of the NOFLOW flag. The updated display data is initialized for later display on the LEDs by the multiplex () display refresh interrupt handler.

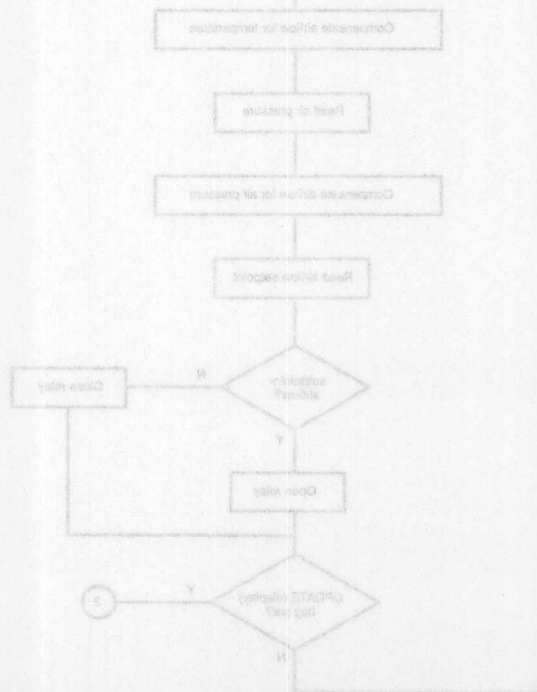
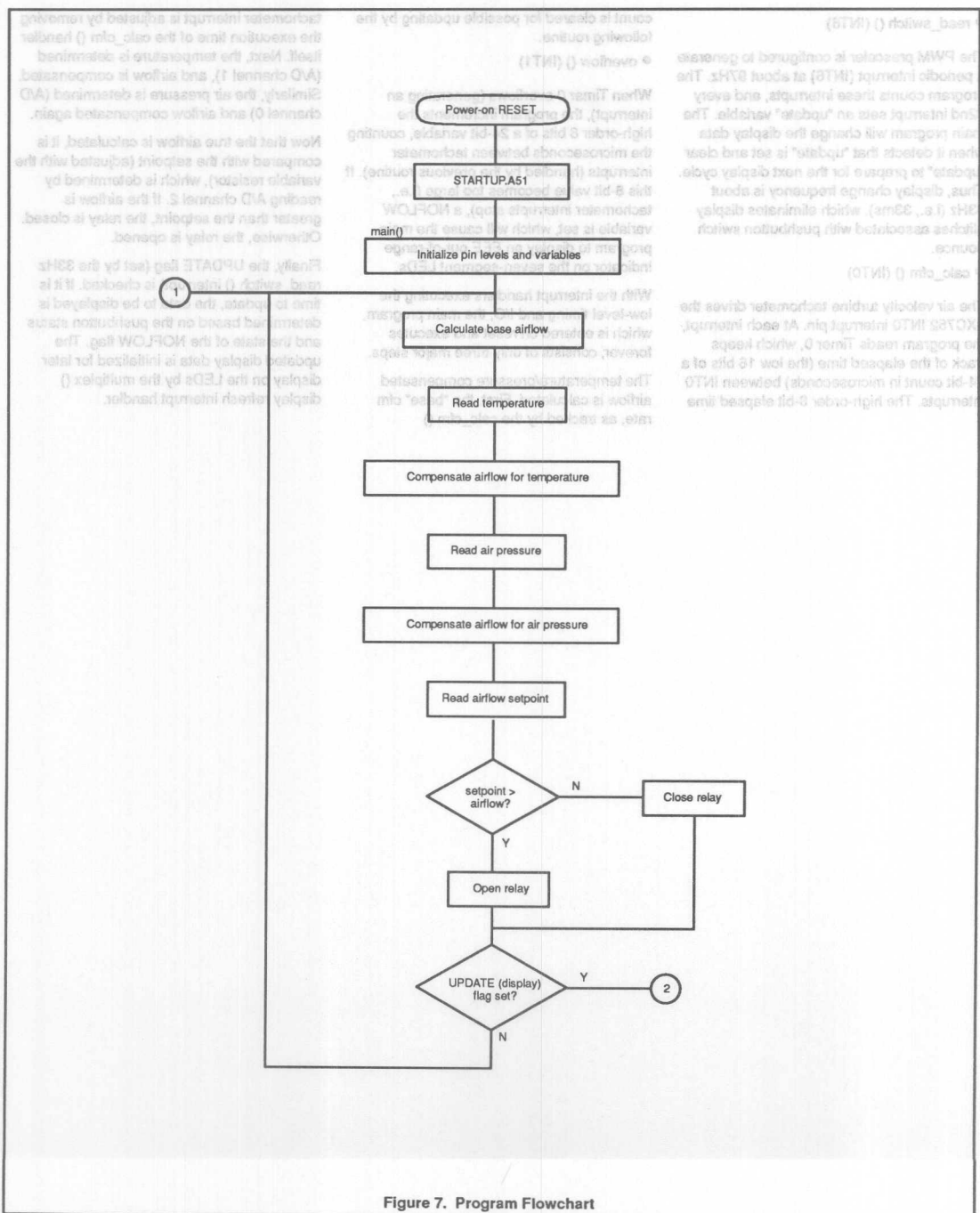


Figure 7. Program Flowchart

Airflow measurement using the 83/87C752 and "C" measurement AN429



Airflow measurement using the 83/87C752 and "C" AN429

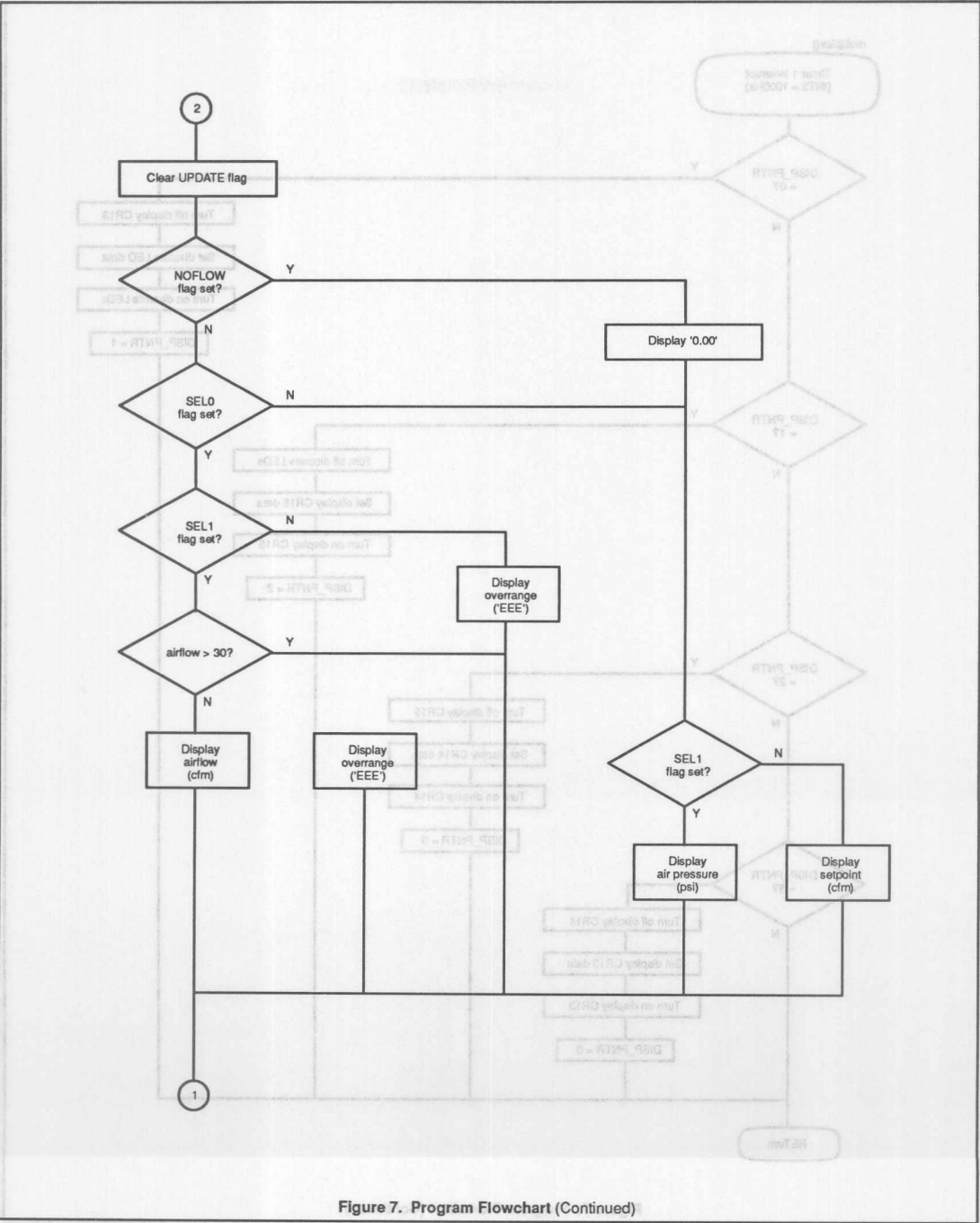
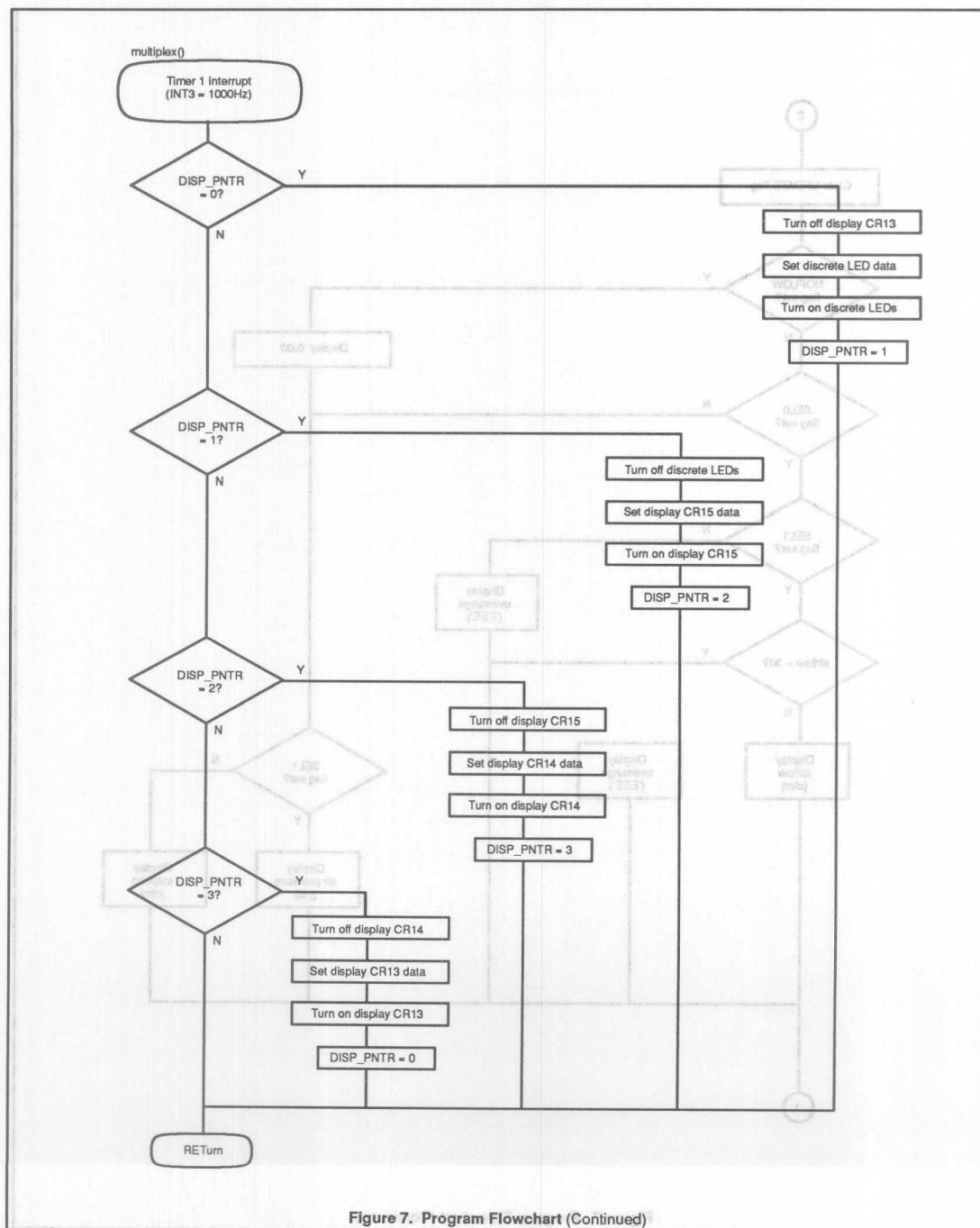


Figure 7. Program Flowchart (Continued)

Airflow measurement using the 83/87C752 and "C" AN429



Airflow measurement using the 83/87C752 and "C" AN429

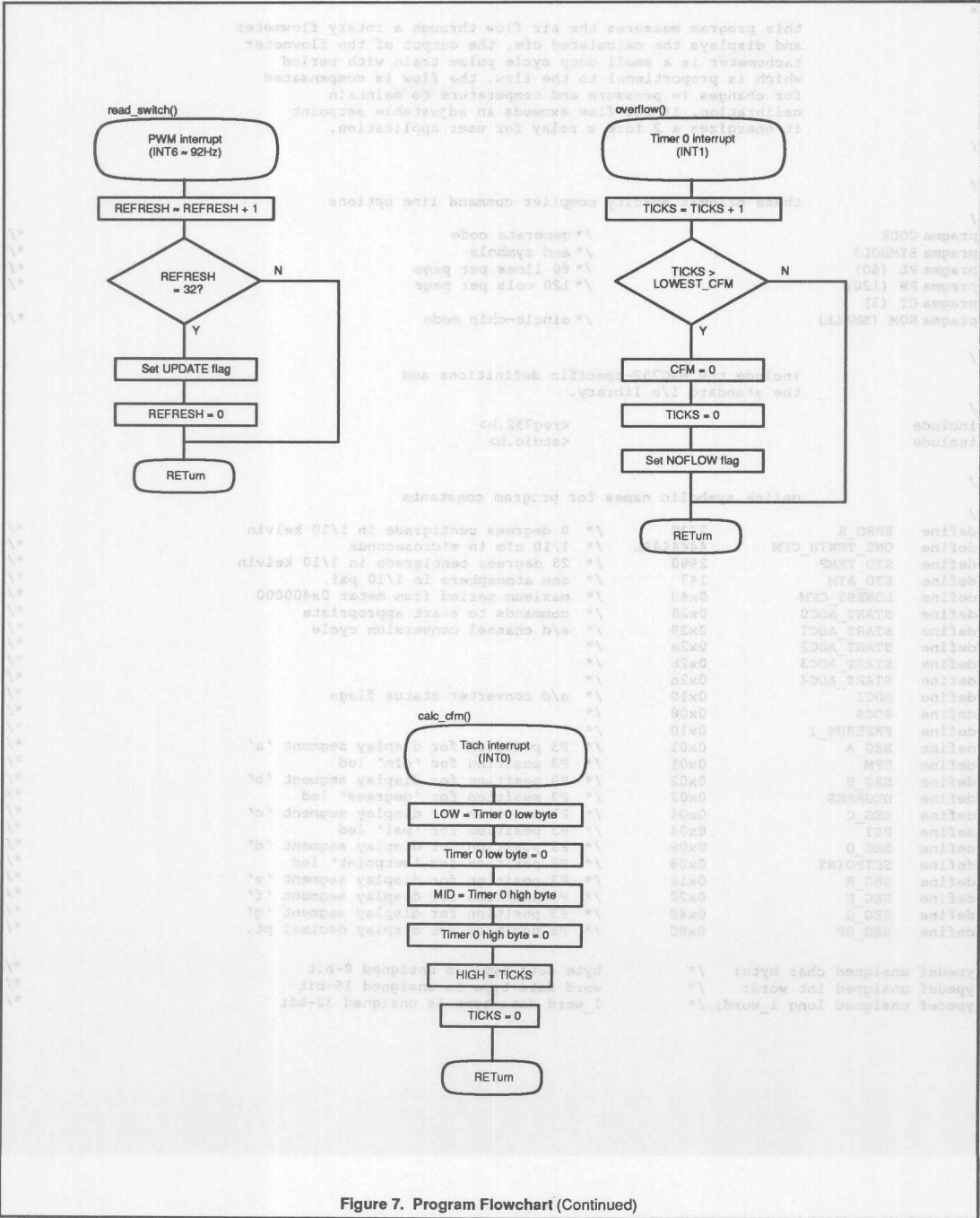


Figure 7. Program Flowchart (Continued)

Airflow measurement using the 83/87C752 and "C" gnu measurement AN429

```

/*
this program measures the air flow through a rotary flowmeter
and displays the calculated cfm. the output of the flowmeter
tachometer is a small duty cycle pulse train with period
which is proportional to the flow. the flow is compensated
for changes in pressure and temperature to maintain
calibration. if the flow exceeds an adjustable setpoint
it energizes a 2 form c relay for user application.

*/

/*
these pragmas specify compiler command line options
*/
#pragma CODE /* generate code
#pragma SYMBOLS /* and symbols
#pragma PL (60) /* 60 lines per page
#pragma PW (120) /* 120 cols per page
#pragma OT (3) /* single-chip mode
#pragma ROM (SMALL)

*/

include the 8XC752-specific definitions and
the standard i/o library.

#include <reg752.h>
#include <stdio.h>

*/

define symbolic names for program constants
*/
#define ZERO_K 2730 /* 0 degrees centigrade in 1/10 kelvin
#define ONE_TENTH_CFM 4444444L /* 1/10 cfm in microseconds
#define STD_TEMP 2980 /* 25 degrees centigrade in 1/10 kelvin
#define STD_ATM 147 /* one atmosphere in 1/10 psi
#define LOWEST_CFM 0x40 /* maximum period from meter 0x400000
#define START_ADC0 0x28 /* commands to start appropriate
#define START_ADC1 0x29 /* a/d channel conversion cycle
#define START_ADC2 0x2a /*
#define START_ADC3 0x2b /*
#define START_ADC4 0x2c /*
#define ADCI 0x10 /* a/d converter status flags
#define ADCS 0x08 /*
#define FREERUN_I 0x10 /*
#define SEG_A 0x01 /* P3 position for display segment 'a'
#define CFM 0x01 /* P3 position for 'cfm' led
#define SEG_B 0x02 /* P3 position for display segment 'b'
#define DEGREES 0x02 /* P3 position for 'degrees' led
#define SEG_C 0x04 /* P3 position for display segment 'c'
#define PSI 0x04 /* P3 position for 'psi' led
#define SEG_D 0x08 /* P3 position for display segment 'd'
#define SETPOINT 0x08 /* P3 position for 'setpoint' led
#define SEG_E 0x10 /* P3 position for display segment 'e'
#define SEG_F 0x20 /* P3 position for display segment 'f'
#define SEG_G 0x40 /* P3 position for display segment 'g'
#define SEG_DP 0x80 /* P3 position for display decimal pt.

typedef unsigned char byte; /*
typedef unsigned int word; /*
typedef unsigned long l_word; /*
byte data type is unsigned 8-bit
word data type is unsigned 16-bit
l_word data type is unsigned 32-bit

```

Figure 7. Program Flowchart (Continued)

Airflow measurement using the 83/87C752 and "C" AN429

```

#define TRUE 1          /* define logical true / false */
#define FALSE 0        /* values for bit variables */

/*
define look-up table of possible seven segment display
characters. the table consists of 11 elements corresponding
to the 10 digits ('0'-'9') and error symbol ('E') that can be
displayed. Each element is defined by ANDing (|) the bit
mask for each segment (SEG_A - SEG_G) comprising the
character. the table contents need to be inverted before
use to be compatible with U2 (udn2585a). for example,
'~segments[3]' specifies the segment mask to display '3'.
*/

code byte segments [ ] =
{
    SEG_A | SEG_B | SEG_C | SEG_D | SEG_E | SEG_F, /* 0 */
    SEG_A | SEG_B | SEG_C, /* 1 */
    SEG_A | SEG_B | SEG_D | SEG_E | SEG_F, /* 2 */
    SEG_A | SEG_B | SEG_C | SEG_D | SEG_E | SEG_F | SEG_G, /* 3 */
    SEG_A | SEG_B | SEG_C | SEG_D | SEG_F | SEG_G, /* 4 */
    SEG_A | SEG_C | SEG_D | SEG_E | SEG_F | SEG_G, /* 5 */
    SEG_A | SEG_B | SEG_C | SEG_D | SEG_E | SEG_F | SEG_G, /* 6 */
    SEG_A | SEG_B | SEG_C, /* 7 */
    SEG_A | SEG_B | SEG_C | SEG_D | SEG_E | SEG_F | SEG_G, /* 8 */
    SEG_A | SEG_B | SEG_C | SEG_D | SEG_F | SEG_G, /* 9 */
    SEG_A | SEG_D | SEG_E | SEG_F | SEG_G, /* E */
};

/*
define the '752 special function bits which control i/o lines.
note that i/o line (and constant) names are capitalized */
sbit RELAY = 0x96; /* active hi to turn on setpoint relay */
sbit STROBE_0 = 0x80; /* active hi to enable display status led's */
sbit STROBE_1 = 0x81; /* active hi to enable display cr15 (tenths) */
sbit STROBE_2 = 0x82; /* active hi to enable display cr14 (ones) */
sbit NO_FLOW = 0x83; /* flag set when no flow detected */
sbit STROBE_3 = 0x84; /* active hi to enable display cr13 (tens) */
sbit SEL_0 = 0x93; /* active low pushbutton inputs used to */
sbit SEL_1 = 0x94; /* select the display mode */
sbit INTR = 0x95; /* */
sbit UPDATE = 0x97; /* flag set when time to update display */

/*
define memory variables. note memory variable names are lower case */
data word cfm; /* gas flow in tenths of a cfm */
data word setpoint; /* relay setpoint in tenths of a cfm */
data word degree_c; /* temperature in tenths centigrade */
data word l_word; /* intermediate calculation value */
data word psi; /* pressure in tenths of a psi */
data byte display0; /* variables to hold values for the */
data byte display1; /* displays during refresh. */
data byte display2; /* display0=status LEDs, display1=CR15, */
data byte display3; /* display2=CR14, display3=CR13 */
data byte disp_ptr; /* pointer to next display to enable */
data byte refresh; /* counter determines display updates */
data byte high; /* bits 16 - 23 of flow period */
data byte middle; /* bits 8 - 15 of flow period */
data byte low; /* bits 0 - 7 of flow period */
data byte ticks; /* incremented by timer overflow */

```

Airflow measurement using the 83/87C752 and "C" AN429

```

/* the program consists of four interrupt handlers (multiplex,
   read switch, overflow, calc_cfm) and a main program.
   multiplex - refresh the seven-segment and discrete status LEDs
   read switch - signal periodic pushbutton sampling and display update
   overflow - accumulate high order bits of time between tach pulses
   calc_cfm - accumulate low order bits of time between tach pulses
   main - calc airflow, control relay, sample pushbuttons, update display
*/

/*
   multiplex
   use the free-running I timer to multiplex the seven-segment and
   discrete leds at approx. 1000 hz.
*/

void multiplex () interrupt 3
{
    switch (disp_ptr)
    {
        case 0x00:
            STROBE_3 = FALSE; /* turn off display crl3 */
            P3 = 0xff; /* turn off all segments */
            P3 = display0; /* load segments for led's */
            STROBE_0 = TRUE; /* turn on status led's */
            disp_ptr = 1; /* increment ptr to display */
            break;
        case 0x01:
            STROBE_0 = FALSE; /* turn off status led's */
            P3 = 0xff; /* turn off all segments */
            P3 = display1; /* load segments for tenths */
            STROBE_1 = TRUE; /* turn on display crl5 */
            disp_ptr = 2; /* increment ptr to display */
            break;
        case 0x02:
            STROBE_1 = FALSE; /* turn off display crl5 */
            P3 = 0xff; /* turn off all segments */
            P3 = display2; /* load segments for units */
            STROBE_2 = TRUE; /* turn on display crl4 */
            disp_ptr = 3; /* increment ptr to display */
            break;
        case 0x03:
            STROBE_2 = FALSE; /* turn off display crl4 */
            P3 = 0xff; /* turn off all segments */
            P3 = display3; /* load segments for tens */
            STROBE_3 = TRUE; /* turn on display crl3 */
            disp_ptr = 0; /* increment ptr to display */
    }
}

```

Airflow measurement using the 83/87C752 and "C" gnuen nemeuassm AN429

```

/*
    read_switch -
    use the free running pwm prescaler to generate
    interrupts at 92 hz. every 32nd interrupt set
    the UPDATE flag which causes main () to sample
    the pushbuttons and update the led displays.
    if the UPDATE flag is set, sample the pushbuttons and update the display data.
*/

void read_switch () interrupt 6
{
    if (refresh++ == 32)
    {
        UPDATE = TRUE;
        refresh = 0;
    }
}

/*
    overflow -
    whenever time0 overflows (from 0xffff to 0x0000)
    increment the variable 'ticks' which accumulates the
    highest order (16 - 23) bits of the gas flow period
    in microseconds. if the variable 'ticks' is greater
    than the period corresponding to a flow of < 0.1 cfm
    then set the NO_FLOW flag which causes main () to
    display '00.0'
*/

void overflow () interrupt 1
{
    if (++ticks > LOWEST_CFM)
    {
        cfm = 0;
        ticks = 0;
        NO_FLOW = TRUE;
    }
}

/*
    calc_cfm -
    an external interrupt (int0) generated by a tach
    pulse from the flowmeter transfers the current value
    of timer0 into variables 'low' and 'middle', and then
    resets the timers. the 'ticks' variable described
    above is also copied to variable 'high', and then
    reset to zero. the NO_FLOW flag is cleared to
    enable display by main () of the calculated cfm.
*/

void calc_cfm () interrupt 0
{
    low = TL0;
    TL0 = 0;
    middle = TH0;
    TH0 = 0;
    high = ticks;
    ticks = 0;
    NO_FLOW = FALSE;
}

```

Airflow measurement using the 83/87C752 and "C" AN429

```

/*
main -
after initializing pins and variables, enter a continuous loop to...
- calculate the airflow based on the tach, temp and pressure inputs.
- compare the airflow to the setpoint input, and control the relay.
- if the UPDATE flag is set (by the read_switch interrupt handler),
  sample the pushbuttons and update the display data.
*/

void main ()
{
    RELAY      = 0;          /* initialize output pins */
    INTR       = 1;
    UPDATE     = 1;
    STROBE_0   = 0;
    STROBE_1   = 0;
    STROBE_2   = 0;
    STROBE_3   = 0;
    NO_FLOW    = 0;
    I2CFG      = FREERUN_I;  /* enable I timer to run, no i2c */
    RTL        = 0;         /* timer 0 period 0x10000 u_seconds */
    RTH        = 0;
    PWMP       = 255;       /* pwm timer interrupt at 923 hz */
    TR         = 1;         /* enable timer 0 */
    IT0        = 1;         /* INT0 is edge active */
    ticks      = 0;         /* initialize variables */
    cfm        = 0;
    low        = 0;
    middle     = 0;
    high       = 0;
    degree_c   = 250;       /* 25.0 tenths degrees c */
    psi        = 147;       /* 14.7 tenths psi */
    corr       = 0;
    refresh    = 0;
    disp_pntr  = 0;
    IE         = 0xab;      /* enable interrupts */

    /*
    main execution loop, executes forever.
    while(1)
    {
        /*
        calculate base cfm rate - first create long word representing
        flow rate period in microseconds. then subtract the time
        overhead in servicing the routine 'calc_cfm'. then divide the
        period into the period for 1/10 cfm, to get flow rate in 1/10
        cfm resolution.

        corr = high * 0x10000L;
        corr += (middle * 0x100L);
        corr += low;
        corr -= CORRECTION;
        corr = ONE_TENTH_CFM / corr;
        */
    }
}

```

Airflow measurement using the 83/87C752 and "C" AN429

```

/*                                     least if cfm rate greater or equal to the
read temperature - measure output from the LM35 sensor,
scaled by the AMP-02. the scaling results in a range
of 0 to 51.0 degrees centigrade, in 0.2 degree steps.
*/
                                     if (setpoint > cfm)
                                     RELAY = 0;
                                     else
                                     RELAY = 1;

ADCON = START_ADC1;
while (ADCON & ADCS) ;
degree_c = ADAT;
degree_c *= 2;

/*                                     test if UPDATE flag has been set, and
compensate cfm rate for temperature - convert temperature
into degrees kelvin, then divide it into the measured flow
rate multiplied by the calibration temperature of the flow-
meter in degrees kelvin. (nominal 25 degrees centigrade)
*/
                                     then test if the NO_FLOW flag has been set. if so then
corr *= STD_TEMP;
corr /= (ZERO_K + degree_c);

/*                                     if NO_FLOW
read pressure - measure output of the KP100A pressure trans-
ducer, scaled by the AMP_02. the scaling results in a range
of 0 to 25.5 psi, in 1/10 psi steps.
*/
                                     displays = -10;
                                     displays = -10;

ADCON = START_ADC0;
while (ADCON & ADCS) ;
psi = ADAT;
if the NO_FLOW flag was not set then read and display the appropriate
select pushbuttons, and display the appropriate data.

/*
compensate cfm rate for pressure - multiply measured pres-
sure and the calculated flow rate, and then divide it by
the standard atmospheric pressure at sea-level. (nominal
14.7 psi)
                                     if (REL_1)
corr *= psi;
corr /= STD_ATM;
cfm = corr;
if no pushbutton is depressed then the default is
the flow rate in cfm. if the flow rate is greater than
or equal to 30 cfm then display the overflow message
'RES'. otherwise,
*/
read setpoint pot to obtain setpoint in the range of
0 - 25.5 cfm in 1/10 cfm steps.

/*
                                     if (cfr => 300)
ADCON = START_ADC2;
while (ADCON & ADCS) ;
setpoint = ADAT;
cfm = 10;
displays = -10;
cfm = 10;
displays = -10;
*/

```

Airflow measurement using the 83/87C752 and "C" AN429

```

test if cfm rate greater or equal to the
setpoint, and if so then energize relay
*/
    if (setpoint > cfm)
        RELAY = 0;
    else
        RELAY = 1;

test if UPDATE flag has been set, and if so reset flag.

if (UPDATE)
    UPDATE = 0;

then test is the NO_FLOW flag has been set. if so then
display '00.0' cfm

    if (NO_FLOW)
    {
        display0 = ~CFM;
        display1 = ~segments[0];
        display2 = ~(segments[0] | SEG_DP);
        display3 = ~segments[0];
    }

if the NO_FLOW flag was not set then read the display
select pushbuttons, and display the appropriate data.

    else if (SEL_0)
    {
        if (SEL_1)
        {
            if no pushbutton is depressed then the default display is
            the flow rate in cfm. if the flowrate is greater than
            or equal to 30 cfm then display the overrange message
            'EEE', otherwise display the flow in 'XX.X' format.

            if (cfm <= 300)
            {
                display0 = ~CFM;
                display1 = ~segments[cfm % 10];
                cfm /= 10;
                display2 = ~(segments[cfm % 10]);
                cfm /= 10;
                display3 = ~segments [cfm % 10];
            }
        }
    }

```


Airflow measurement using the 83/87C752 and "C"

AN429

With these integrated functions that the 83/87C752 offers, and the ability to provide a complete solution to power resource control, this device is emerging to be the industry standard for power management.

Figure 1 shows a block diagram of a typical system implementation. It employs the integrated power management scheme via keyboard and power management function. The CPU and coprocessor reside on the local bus with the system memory. A local bus controller monitors CPU bus cycles to see if they are memory or ISA cycles. It also monitors the interrupt and DMA functions. The ISA bus controller processes non-system memory bus cycles. A frequency generator is used to provide the system clocks and clock monitoring. The peripheral controller integrates the communications and mass storage sub-system. The VGA sub-system stores and displays the VGA information.

Figure 2 shows the microcontroller with the external support devices. The frequency generator provides the system clocks. It must have the ability to change the frequency of the clocks without losing the minimum high or low times for the core logic. The frequency generator can control the speed of the system clocks via frequency select pins on the frequency generator. Frequency generators such as the Avram AV6157 can change the processor clock speed gradually and continuously without affecting the minimum high or low times.

The integrated controller monitors system activity via the digital input pins. It uses internal timers to time the intervals between activity. The power to the VGA and peripheral sub-systems is controlled by the digital output pins via MOSFETs.

Battery level and VCC is monitored by the onboard A/D converter.

Figure 3 shows the power management software. The power management software is implemented in the 83/87C752.

The power management software is implemented in the 83/87C752.

The power management software is implemented in the 83/87C752.

The power management software is implemented in the 83/87C752.

The power management software is implemented in the 83/87C752.

The power management software is implemented in the 83/87C752.

The power management software is implemented in the 83/87C752.

The power management software is implemented in the 83/87C752.

The power management software is implemented in the 83/87C752.

The power management software is implemented in the 83/87C752.

The power management software is implemented in the 83/87C752.

The power management software is implemented in the 83/87C752.

The power management software is implemented in the 83/87C752.

The power management software is implemented in the 83/87C752.

The power management software is implemented in the 83/87C752.

else
display0 = ~CFM;
display1 = ~segments[10];
display2 = ~segments[10];
display3 = ~segments[10];
}

if the temp pushbutton (SW1) is pressed then display the air temperature.

if the temp pushbutton (SW1) is pressed then display the air temperature.

if the temp pushbutton (SW1) is pressed then display the air temperature.

if the temp pushbutton (SW1) is pressed then display the air temperature.

if the temp pushbutton (SW1) is pressed then display the air temperature.

if the temp pushbutton (SW1) is pressed then display the air temperature.

if the temp pushbutton (SW1) is pressed then display the air temperature.

if the temp pushbutton (SW1) is pressed then display the air temperature.

if the temp pushbutton (SW1) is pressed then display the air temperature.

if the temp pushbutton (SW1) is pressed then display the air temperature.

if the temp pushbutton (SW1) is pressed then display the air temperature.

if the temp pushbutton (SW1) is pressed then display the air temperature.

if the temp pushbutton (SW1) is pressed then display the air temperature.

if the temp pushbutton (SW1) is pressed then display the air temperature.

if the temp pushbutton (SW1) is pressed then display the air temperature.

if the temp pushbutton (SW1) is pressed then display the air temperature.

if the temp pushbutton (SW1) is pressed then display the air temperature.

if the temp pushbutton (SW1) is pressed then display the air temperature.

if the temp pushbutton (SW1) is pressed then display the air temperature.

if the temp pushbutton (SW1) is pressed then display the air temperature.

if the temp pushbutton (SW1) is pressed then display the air temperature.

if the temp pushbutton (SW1) is pressed then display the air temperature.

if the temp pushbutton (SW1) is pressed then display the air temperature.

if the temp pushbutton (SW1) is pressed then display the air temperature.

With conventional microprocessor based systems, the method was primarily concerned with performance, cost and features.

With the advent of hand-held and portable computers, the prominent market requirements focus on size, weight and battery life.

Given a mature 386/486 architecture that provides more than adequate performance for various notebook applications usage, the design challenges for these machines involve around developing low power systems that maintain battery usage.

The features of a notebook PC are: lightweight, portable, and battery life. The lowest component cost is usually the battery and the choice of battery by the power designer. Thus the performance of the power management scheme has a direct bearing on both these parameters.

The battery life for notebook machines is around 3 hours (in the US market) from coast to coast (USA).

The battery life for notebook machines is around 3 hours (in the US market) from coast to coast (USA).

The battery life for notebook machines is around 3 hours (in the US market) from coast to coast (USA).

The battery life for notebook machines is around 3 hours (in the US market) from coast to coast (USA).

The battery life for notebook machines is around 3 hours (in the US market) from coast to coast (USA).

The battery life for notebook machines is around 3 hours (in the US market) from coast to coast (USA).

The battery life for notebook machines is around 3 hours (in the US market) from coast to coast (USA).

The battery life for notebook machines is around 3 hours (in the US market) from coast to coast (USA).

The battery life for notebook machines is around 3 hours (in the US market) from coast to coast (USA).

The battery life for notebook machines is around 3 hours (in the US market) from coast to coast (USA).

The battery life for notebook machines is around 3 hours (in the US market) from coast to coast (USA).

The battery life for notebook machines is around 3 hours (in the US market) from coast to coast (USA).

The battery life for notebook machines is around 3 hours (in the US market) from coast to coast (USA).

The battery life for notebook machines is around 3 hours (in the US market) from coast to coast (USA).

The battery life for notebook machines is around 3 hours (in the US market) from coast to coast (USA).

The battery life for notebook machines is around 3 hours (in the US market) from coast to coast (USA).

The battery life for notebook machines is around 3 hours (in the US market) from coast to coast (USA).

The battery life for notebook machines is around 3 hours (in the US market) from coast to coast (USA).

The battery life for notebook machines is around 3 hours (in the US market) from coast to coast (USA).

The battery life for notebook machines is around 3 hours (in the US market) from coast to coast (USA).

The battery life for notebook machines is around 3 hours (in the US market) from coast to coast (USA).

The battery life for notebook machines is around 3 hours (in the US market) from coast to coast (USA).

"Opti-Mizer" power management for notebook computers using the 8XC752 microcontroller

AN436

INTRODUCTION

With conventional microprocessor based systems, the market was primarily concerned with performance, cost and features. With the advent of hand-held and portable computers, the prominent market requirements focus on size, weight and battery life.

Given a mature 386SX/AT architecture that provides more than adequate performance for average notebook application usage, the design challenges for these machines revolve around developing low power systems that maximize battery usage.

The features of a notebook PC are usually characterized as weight and battery life. The heaviest component of a PC is usually the battery and the choice of battery is dictated by the power required. Thus the performance of the power management scheme has a direct bearing on both these parameters.

The battery life targets for notebook machines is around 5 hours (the air commute time from coast to coast USA).

OVERVIEW

Most of the power consumed by a fully powered PC is wasted. The hard disk spins constantly even though data transfers from the disk are very sporadic. PCs may sit unattended for periods where the user is distracted by a telephone call, for instance.

There are two principle challenges in designing a power management system: the ability to power down various devices without affecting other devices on the same bus, and ensuring full compatibility with existing operating systems and applications.

The largest user of power in a PC is the display sub-system (3-5 Watts) followed by the peripherals such as the hard disk (2-4 Watts), the main system memory (0.5-1.5 Watts), and the core logic (1 Watt).

INTEGRATED POWER MANAGEMENT

The conventional PC architecture needs to be extended to support power management. Hardware needs to be added to provide power-down capabilities and software needs to be added to support the hardware and provide DOS compatibility. The software support is usually realized in the BIOS. The hardware support can be implemented with external circuitry. This external circuitry manages the power resources to individual sub-sections of the PC system as these

resources dictate. The external circuitry monitors battery power, system activities and timed events.

Conventionally, the external circuitry is comprised of a digital power management ASIC and associated components. The use of this part increases the chip count of the PC system.

The power management system has to determine how the system resources are being used. The resource usage of a PC can be determined by monitoring events or activities. User activity is usually determined by monitoring the keyboard controller for keystroke events. Keystroke events can be indicated by interrupts to the PC core logic or IO reads to the keyboard controller location.

The power management ASIC solutions on the market today, such as the VADEM/INTEL 82C347, VLSI VL82C312 and INTEL 80C386SL all require external analog support circuitry to completely implement the power management functions. For example, low battery detect is implemented by the use of external comparator chains and complex, close tolerance level detect circuitry. The cost of this external circuitry is usually a significant proportion of the overall cost of the power management solution. The 83/87C752 employs an internal analog-to-digital converter (ADC). The ADC can be used to implement the battery level detection function at no extra cost and with no extra support circuitry.

The 83/87C752 is a member of the Philips 8051 family of high performance 8-bit microcontrollers. These processors have been optimized for sequential real time control applications. The 83/87C752 contains most of the features of the 80C51 and has the following features:

- 2k bytes ROM
- 64 bytes RAM
- Single level interrupt structure
- 16 bit programmable counter/timer
- Two 8-bit and one 5-bit bi-directional IO ports
- I²C serial interface
- PWM with interrupt and overflow capability
- 5 channels of 8-bit A/D
- 28-pin DIP and PLCC.

FLEXIBILITY

ASIC solutions to power management offer rigid schemes which work adequately with a few notebook architectures, but rarely offer

exactly what the designer requires. With the current competitive arena for laptop development, time-to-market and value added features have a significant impact on the sales success of a particular product. The 83/87C752 offers flexibility at a low price, the power management design requirements can be coded and configured in the controller software and One Time Programmable (OTP) devices can offer a quick low-cost implementation of the coded scheme.

With these integrated functions that the 83/87C752 offers, and its ability to provide a complete solution to power resource control, this device is emerging to be the industry standard for power management.

TOPOLOGY

Figure 1 shows a block diagram of a typical system implementation. It employs the integrated power management scheme using the Philips microcontroller to handle the keyboard and power management functions. The CPU and coprocessor reside on the local bus with the system memory. A local bus controller monitors CPU bus cycles to see if they are memory or ISA cycles. It also integrates the interrupt and DMA functions. The ISA bus controller processes non-system memory bus cycles. A frequency generator is used to provide the system clocks and clock multiplexing. The peripheral controller integrates the communications and mass storage sub-system. The VGA sub-system shares the ISA bus with the peripheral controller. The VGA controller has associated VGA memory.

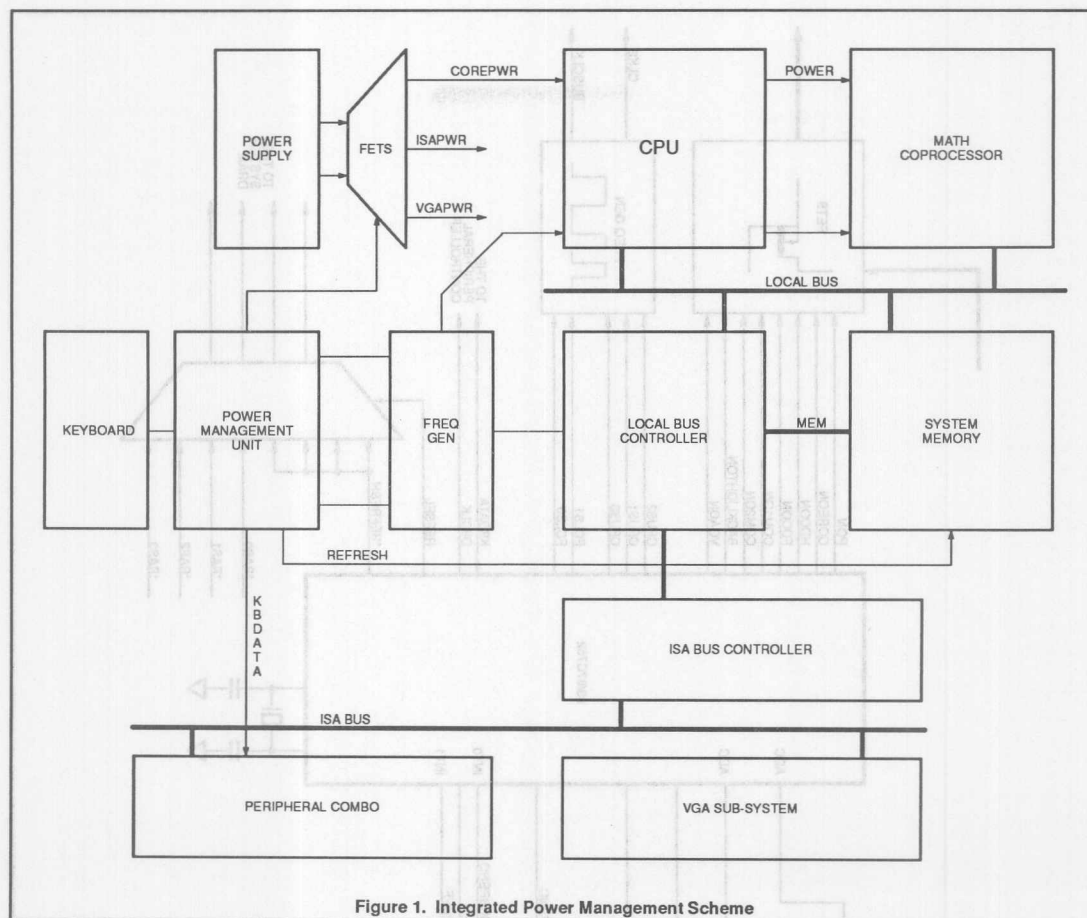
Figure 2 shows the microcontroller with the external support devices. The frequency generator provides the system clocks. It must have the ability to change the frequency of the clocks without violating the minimum high or low times for the core logic. The Philips microcontroller can control the speed of the system clocks via frequency select pins on the frequency generator. Frequency generators such as the Avasem AV9127 can change the processor clock speed gradually and continuously without violating the minimum high or low times.

The integrated controller monitors system activity via its digital input ports. It uses internal timers to time the intervals between activity. The power to the VGA and peripheral sub-systems is controlled by the digital output port pins via MOSFETs.

Battery level and V_{CC} is monitored by the onboard A/D converter.

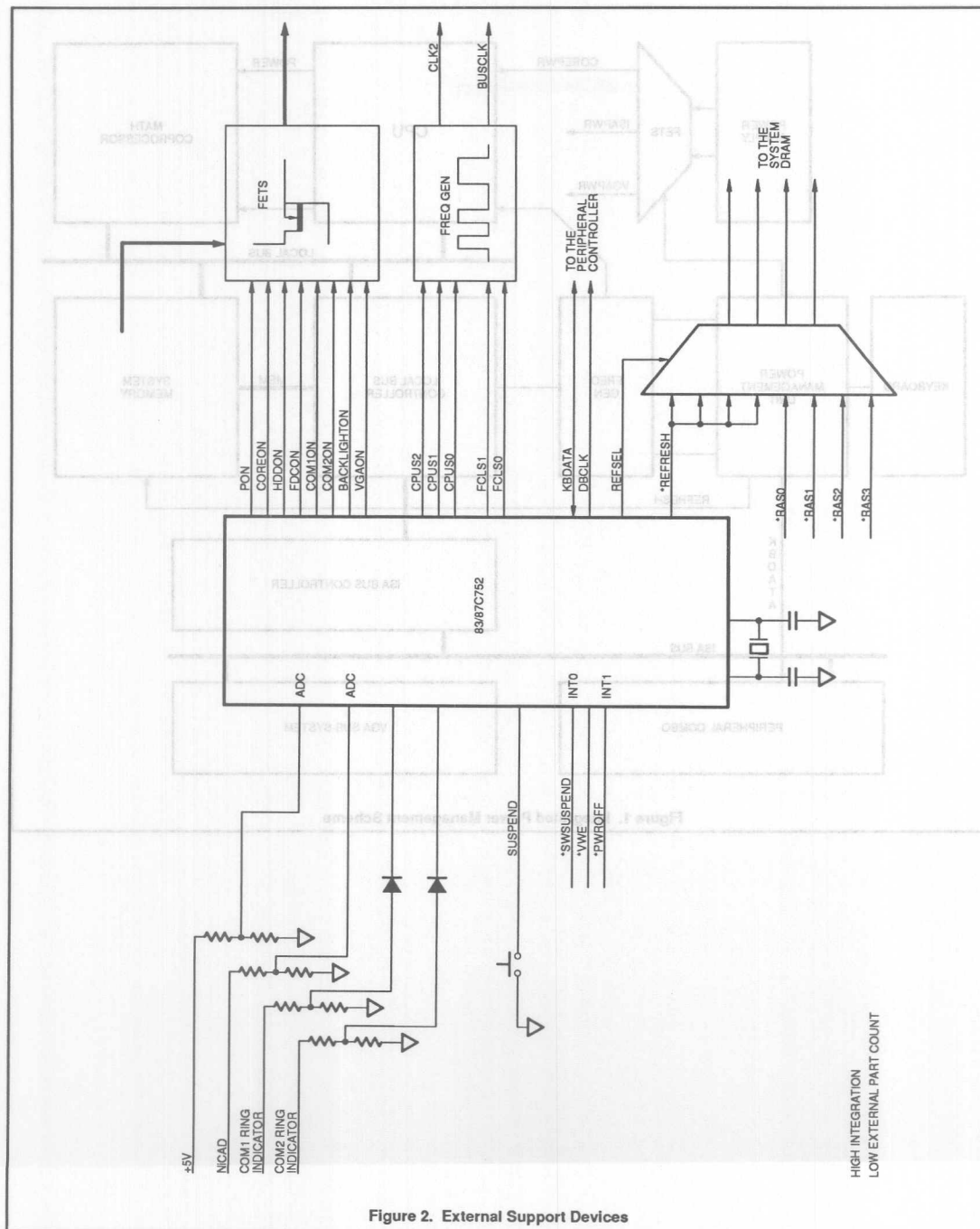
"Opti-Mizer" power management for notebook computers using the 8XC752 microcontroller

AN436



"Opti-Mizer" power management for notebook computers using the 8XC752 microcontroller

AN436



"Opti-Mizer" power management for notebook computers using the 8XC752 microcontroller

AN436

OPERATION

The power management system operates like a state machine. Transitions from state to state are controlled by expiring timers which are retriggered by external events. On entering a state, external power is switched or clocks are modified. A typical power management system would employ six states: Full Power, Doze, Shutdown, Sleep, Suspend, and Off.

The state diagram of Figure 3 shows the power management states and their interrelationships.

Full Power

Entry to this state is controlled by a transition of the ON/OFF switch. In this state all the power control outputs are asserted, and the clock generator is selected for the highest speed. The system runs at full speed and power.

Doze

This state is entered from the FULL POWER state. Entry into this state is controlled by an expired timer (typically 30 secs). The timer expired as a result of not being reloaded by a transition on an activity monitor input pin. In this state the frequency generator is instructed to reduce the clock speed to about half that of the previous state.

Shutdown

This state is also entered from the FULL POWER state and operates in parallel with the DOZE state. Entry into this state is controlled by an expired timer (typically 30 secs). The timer expired as a result of not being reloaded by a transition on an activity monitor pin. In this state the power to a particular peripheral or group of peripherals is removed via an external FET.

Shutdown-Doze

This is an intermediate state which implements the features of both the SHUTDOWN and DOZE states. Entry into this state from the DOZE state is controlled by an expired timer (typically 30 secs). The timer expired as a result of not being reloaded by a transition on an activity monitor pin. Entry into this state from the SHUTDOWN state is controlled by an expired timer also. The timer expired as a result of not being reloaded by a transition on an activity monitor input pin. In this state the power to a particular peripheral or group of peripherals is removed via an external FET and the frequency generator is instructed to reduce the clock speed to about half that of the FULL POWER state.

Sleep

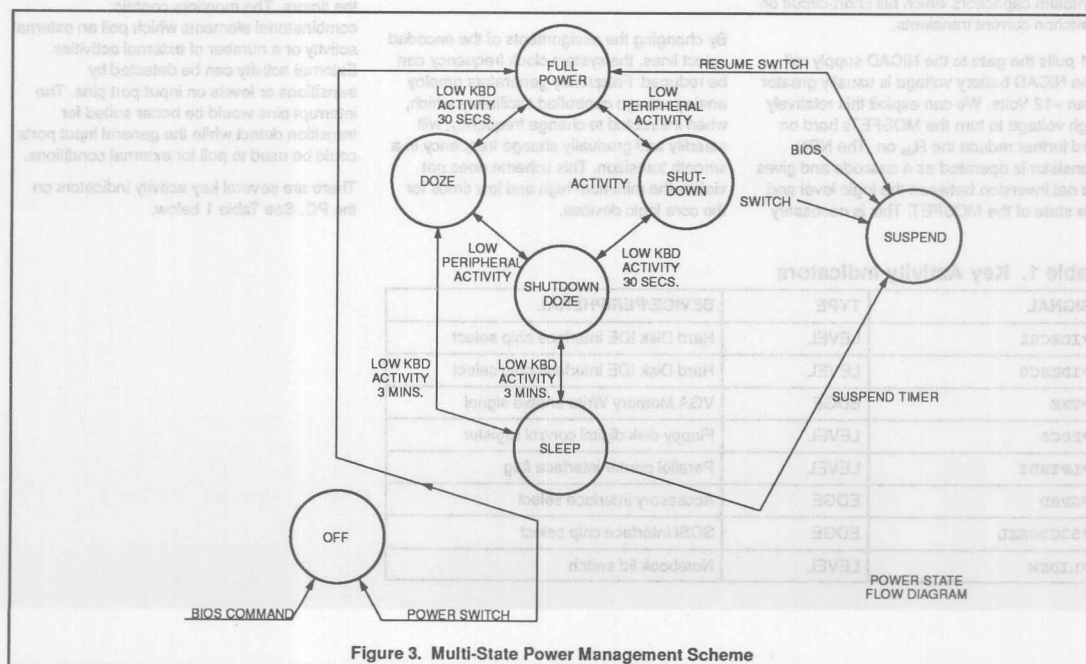
This state is entered from either the DOZE state or SHUTDOWN state. Entry into this state is controlled by an expired timer (typically 30 secs). The length of this timer is usually longer than that employed in the DOZE or SHUTDOWN states. The timer expired as a result of not being reloaded by a transition on an activity monitor pin. The activity monitor may look for keystrokes or video activity as described below. In this state power is removed from the backlight and LCD modulation voltage regulator via external FETs.

Suspend

This state is entered from any of the above states. Entry into this state is controlled by a transition on an external suspend switch or a command from the BIOS. During this state, the microcontroller takes over the task of refreshing the system memory and removes the power from the rest of the system via external FETs.

Off

This state is entered from any of the above states. Entry into this state is controlled by a transition on an external switch or a command from the BIOS.



"Opti-Mizer" power management for notebook computers using the 8XC752 microcontroller

AN436

POWER MANAGEMENT ELEMENTS

Figure 4 shows the internal power management elements of the Philips controller. The Activity monitors contain combinatorial algorithms to monitor or poll an activity or combination of activities. The activity monitors reload programmable timers which toggle clock control and power control output pins.

Each functional block is discussed in more detail below.

Power Outputs

The power outputs are logic level signals that control MOSFETs via level shifting circuitry.

The MOSFETs switch power to the various blocks under power management control and are chosen to have a low R_{ds} on which reduces the voltage drop across the DRAIN-SOURCE channel.

The level shifting circuitry and MOSFET orientation is shown in Figure 5.

C1 and R1 control the switchon edge and should be chosen to make the edge sufficiently slow to minimize the inrush current. This is necessary where the notebook computer employs solid chip tantalum capacitors which fail short-circuit on switchon current transients.

R1 pulls the gate to the NiCAD supply rail. The NiCAD battery voltage is usually greater than +12 Volts. We can exploit this relatively high voltage to turn the MOSFETs hard on and further reduce the R_{ds} on. The NPN transistor is operated as a cascode and gives no net inversion between the logic level and the state of the MOSFET. This is necessary

to handle the default powerup mode of the port pins.

For a lower performance and cost reduced system, a logic level FET can be used such as a MTM25N06L and driven directly from the microcontroller port. These logic level FETs usually have R_{ds} on specifications in the region of 100 milliohms. The finite drain-source resistance implies that a small amount of power is wasted in this channel while power is applied to the switched group of devices.

Clock Control

The clock control module controls the speed of the system clocks. Where the notebook system has a synchronous ISA clock, it is derived directly from CLK2, the processor clock. The clock control outputs can be fed directly to a frequency generator such as an AVASEM AV9127 where pins are committed to encode a frequency select scheme. The scheme employs two programmable clock generators; one with eight preset frequencies used for the system clock; and the other with four preset frequencies used for the mass storage subsystem. Figure 6 shows the interconnection between the clock control port and frequency generator.

By changing the assignments of the encoded select lines, the system clock frequency can be reduced. Frequency generators employ analog voltage controlled oscillators which, when instructed to change frequency, will steadily and gradually change frequency in a smooth transition. This scheme does not violate the minimum high and low times for the core logic devices.

Timers

The timers should run independently of the keyboard scanning function. The timers are used as timeouts for a combination of external events or activities. The timers are constructed of reloadable timers and reloaded by transitions or conditions on external events. A typical timeout period is between 1 and 4 minutes, therefore the timeout must be constructed from both timer hardware and support software. Figure 7 shows the interrelationships between hardware and software. The software must record the instances of timeout cycles. If the number of cycles is allowed to reach a predetermined number, the timeout elapses and the assigned power control output is negated, or the clock control outputs proceed to the next state. The count of the number of cycles is reset by a command for the activity monitor.

The activity monitor asserts flags during the background and interrupt tasks. The timer software processes these flags to determine the state of the timeout. The software uses a count variable to measure the instances of the timer elapsing and a flag to determine whether activity has occurred.

Activity Monitor

The activity monitor sets the activity flags for the timers. The monitors contain combinatorial elements which poll an external activity or a number of external activities. External activity can be detected by transitions or levels on input port pins. The interrupt pins would be better suited for transition detect while the general input ports could be used to poll for external conditions.

There are several key activity indicators on the PC. See Table 1 below.

Table 1. Key Activity Indicators

SIGNAL	TYPE	DEVICE/PERIPHERAL
*IDECSEL	LEVEL	Hard Disk IDE interface chip select
*IDESC0	LEVEL	Hard Disk IDE interface chip select
*VWE	EDGE	VGA Memory Write enable signal
*FDCS	LEVEL	Floppy disk digital control register
*LPTRDY	LEVEL	Parallel printer interface flag
*GPRD	EDGE	Accessory interface select
*53C90SEL	EDGE	SCSI interface chip select
*LIDSW	LEVEL	Notebook lid switch

“Opti-Mizer” power management for notebook computers using the 8XC752 microcontroller

AN436

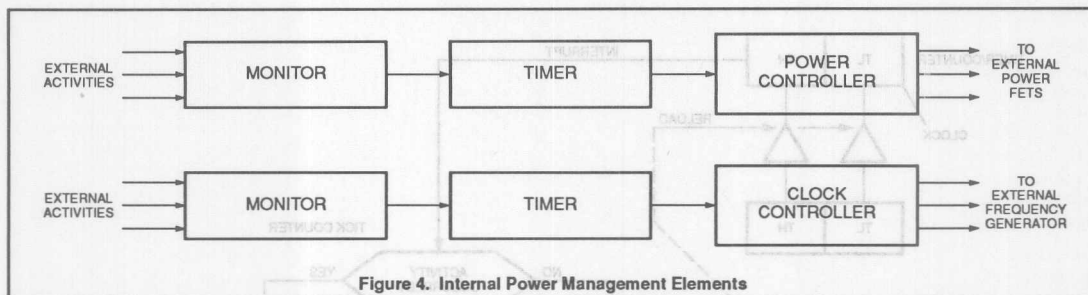


Figure 4. Internal Power Management Elements

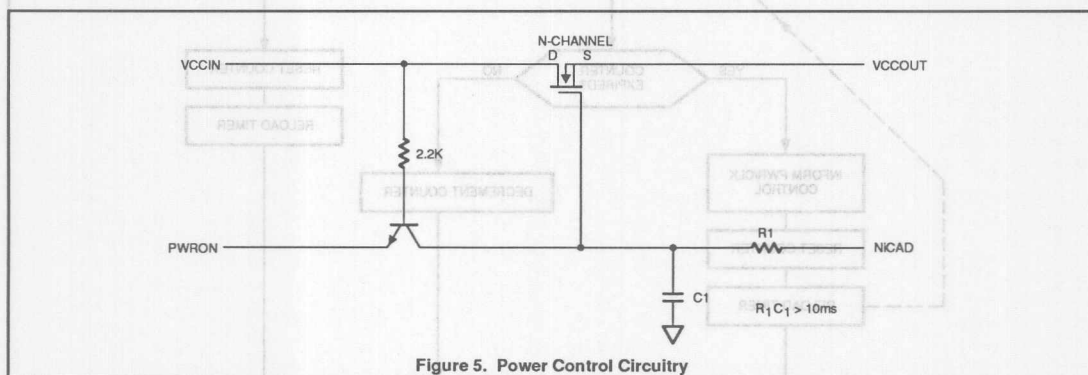


Figure 5. Power Control Circuitry

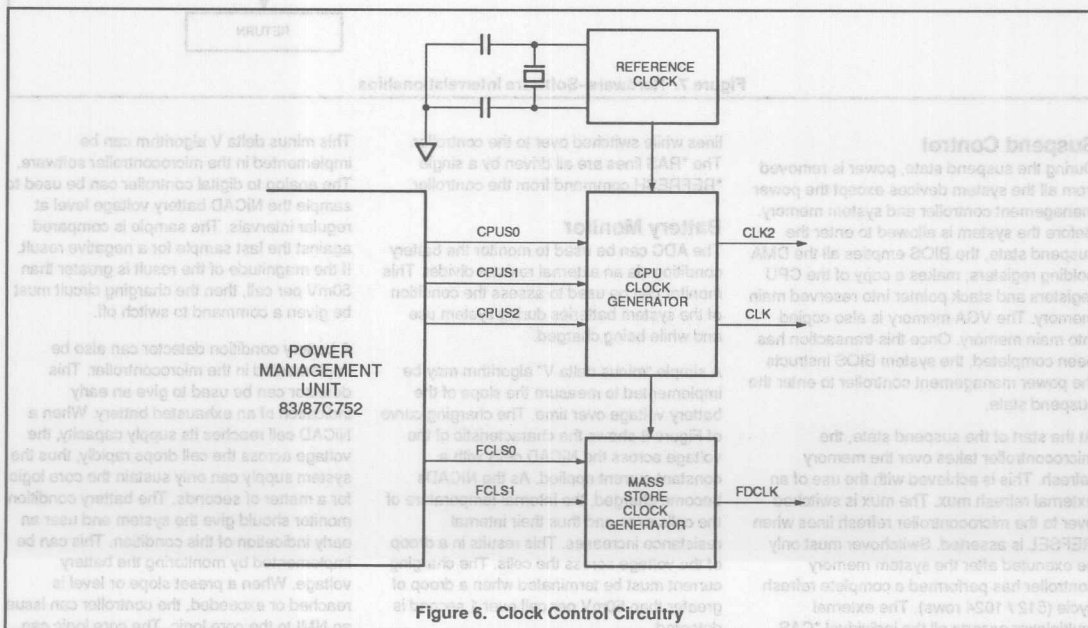


Figure 6. Clock Control Circuitry

“Opti-Mizer” power management for notebook computers using the 8XC752 microcontroller

AN436

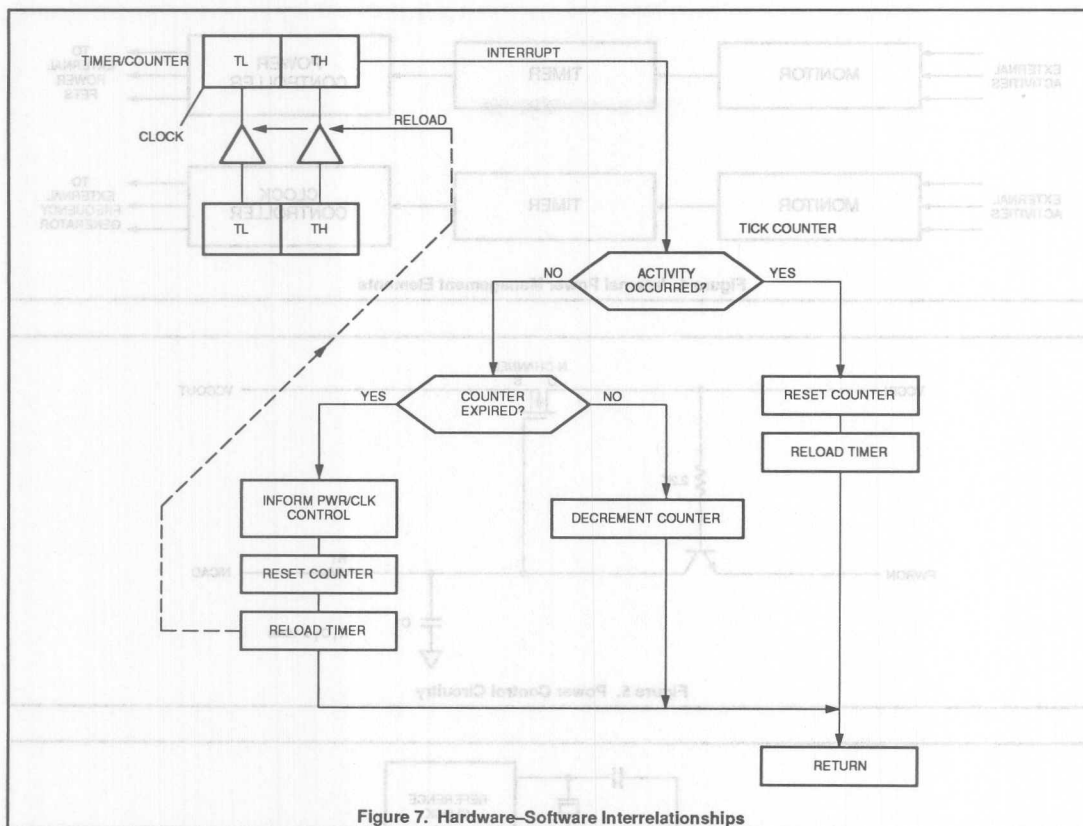


Figure 7. Hardware-Software Interrelationships

Suspend Control

During the suspend state, power is removed from all the system devices except the power management controller and system memory. Before the system is allowed to enter the suspend state, the BIOS empties all the DMA holding registers, makes a copy of the CPU registers and stack pointer into reserved main memory. The VGA memory is also copied into main memory. Once this transaction has been completed, the system BIOS instructs the power management controller to enter the suspend state.

At the start of the suspend state, the microcontroller takes over the memory refresh. This is achieved with the use of an external refresh mux. The mux is switched over to the microcontroller refresh lines when REFSEL is asserted. Switchover must only be executed after the system memory controller has performed a complete refresh cycle (512 / 1024 rows). The external multiplexer asserts all the individual *CAS

lines while switched over to the controller. The *RAS lines are all driven by a single *REFRESH command from the controller.

Battery Monitor

The ADC can be used to monitor the battery condition via an external resistor divider. This monitor can be used to assess the condition of the system batteries during system use and while being charged.

A simple "minus delta V" algorithm may be implemented to measure the slope of the battery voltage over time. The charging curve of Figure 8 shows the characteristic of the voltage across the NiCAD cells with a constant current applied. As the NiCADs become charged, the internal temperature of the cells rises and thus their internal resistance increases. This results in a droop of the voltage across the cells. The charging current must be terminated when a droop of greater than 50mV per cell over 1 second is detected.

This minus delta V algorithm can be implemented in the microcontroller software. The analog to digital controller can be used to sample the NiCAD battery voltage level at regular intervals. The sample is compared against the last sample for a negative result. If the magnitude of the result is greater than 50mV per cell, then the charging circuit must be given a command to switch off.

A battery condition detector can also be implemented in the microcontroller. This detector can be used to give an early indication of an exhausted battery. When a NiCAD cell reaches its supply capacity, the voltage across the cell drops rapidly, thus the system supply can only sustain the core logic for a matter of seconds. The battery condition monitor should give the system and user an early indication of this condition. This can be implemented by monitoring the battery voltage. When a preset slope or level is reached or exceeded, the controller can issue an NMI to the core logic. The core logic can

“Opti-Mizer” power management for notebook computers using the 8XC752 microcontroller

AN436

execute a shutdown routine which saves the state of the machine on the hard disk and preserves the integrity of the user's data.

PERFORMANCE

Table 2 shows the power performance of the 83/87C752 in a typical notebook system. During each power management state, the

core logic system and peripherals demand lower and lower power as the clock speeds are reduced, resets are asserted and power is removed from the device or peripheral.

During the suspend mode, power is completely removed from the core logic devices and peripherals, only the DRAMs remain energized so that the system state

can be restored to the next post suspend cycle on the detection of an activity.

The FULL-ON figures are maximums. In reality, the processor executes sporadic scripts and then idles in tight loops awaiting IO. This means that the actual power required by the system under normal conditions will be lower than that estimated in that column of the table.

Table 2.

STATE DEVICE	FULL-ON POWER (W) (MAX)	SHUTDOWN-DOZE (W)	SLEEP POWER (W)	SUSPEND POWER (W)
Am386SX	2.14	0.7	0.5	0
DRAM + Controller	1.7	0.6	0.4	0.025
Local bus controller	0.3	0.3	0.2	0
ISA bus controller	0.2	0.18	0.07	0
87C752	0.08	0.08	0.08	0.015
BIOS ROM	0.11	0.002	0.002	0
Floppy drive	3.1	0.035	0.035	0
IDE drive	3.3	0.68	0.68	0
VGA controller	2.1	1.6	1.2	0.015
Keyboard controller	0.7	0.6	0.4	0
Keyboard	0.15	0.15	0.15	0
Oscillators	0.1	0.1	0.1	0
RS232 buffers	0.11	0.002	0.002	0
LCD panel	0.7	0.62	0.2	0
Backlight	1.5	1.5	0.1	0
TOTALS	15.69	6.709	4.119	0.055

The gradient of power performance across the states is quite steep, especially when transitioning from the sleep to the suspend states.

Table 3.

	FULL-ON	¹ / ₂ CLK2 SHUTDOWN-DOZE	¹ / ₄ CLK2 SLEEP	SUSPEND
CPU	ON	ON	ON	OFF
DRAM	ON	ON	ON	ON
Local bus	ON	ON	ON	OFF
ISA bus	ON	ON	OFF	OFF
87C752	ON	ON	ON	ON
BIOS ROM	ON	OFF	OFF	OFF
Floppy	ON	ON	OFF	OFF
IDE drive	ON	IDLE	OFF	OFF
VGA	ON	ON	ON	OFF
Keyboard controller	ON	ON	ON	ON
Keyboard	ON	ON	ON	OFF
Oscillators	ON	ON	ON	OFF
RS232 buffers	ON	OFF	OFF	OFF
LCD panel	ON	ON	OFF	OFF
Backlight	ON	ON	OFF	OFF

Note that the panel and backlight are off during the sleep state.

“Opti-Mizer” power management for notebook computers using the 8XC752 microcontroller

AN436

OTHER INTEGRATION OPPORTUNITIES

The 83/87C752 offers a complete power management solution as described above. The functionality and IO capability of this device can be used as a common denominator for other, larger Philips microcontrollers. The 83/87C552 offers the same internal functionality while providing more integration capabilities with its increased IO, memory and timer functions.

An example of an integration opportunity would be to combine the keyboard scanner function with the power management

function. The Philips 83/87C552 microcontroller is ideal for this task. The microcontroller can implement the scanning and code generation schemes associated with the keyboard function, implement the activity monitors, timers and power control scheme required for power management as well as provide an integrated solution to battery condition detection by exploiting the onboard A/D converters.

The PC keyboard scanner is traditionally an 8051 microcontroller. The keyboard scanning and code generation can be performed by an 8051 running at 6MHz. A 12 or 16MHz

83/87C552 microcontroller would have the bandwidth to take on other tasks.

Figure 9 shows the 83/87C552 integrated as a power management unit and keyboard scanner.

Other members of the Philips family of 8051 derivative microcontrollers can also provide an integrated solution to power management (see Table 4).

As can be seen, the 83/87C550 provides an intermediate solution to the 83/87C552 and 83/87C752. It has more memory and IO than the 83/87C752 and has three more ADC channels.

Table 4. Microcontrollers with A/D

DEVICE	ROM	RAM	A/D	I/O	PWM	TIMERS
83/87C550	4k	128	8 8-bit	24	0	2
83/87C552	8k	256	8 10-bit	48	2	3
83/87C752	2k	64	5 8-bit	21	1	1

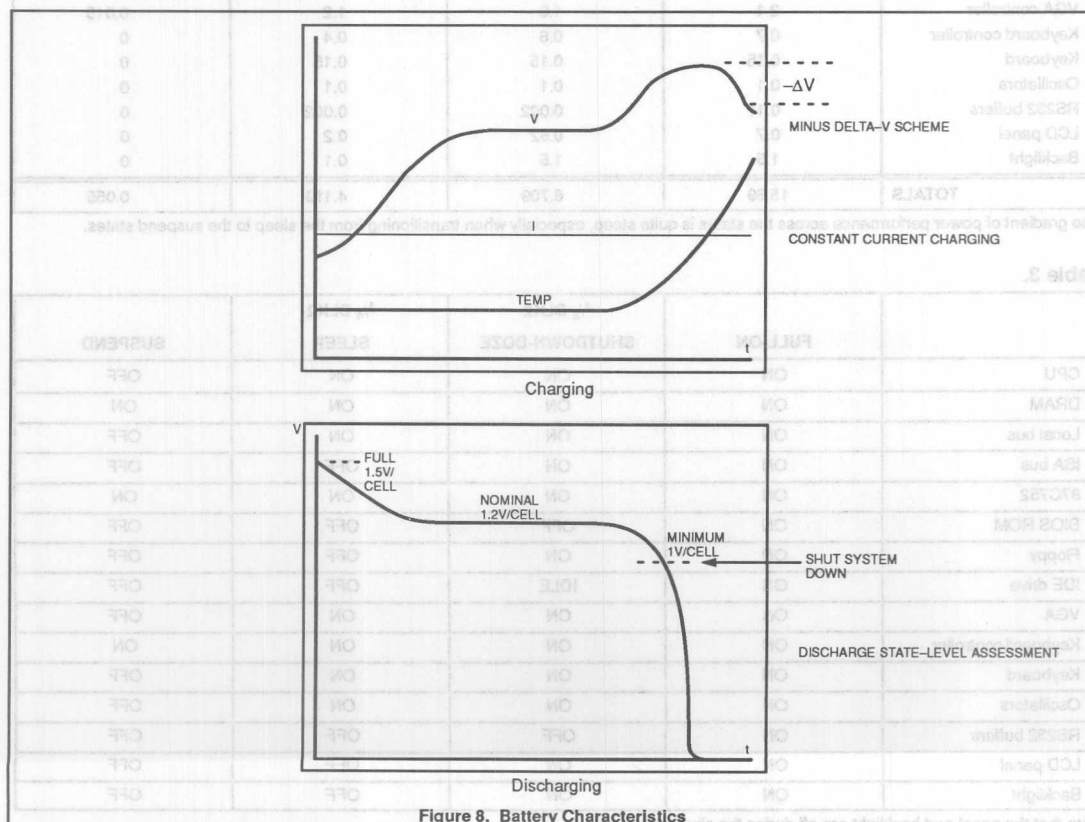


Figure 8. Battery Characteristics

"Opti-Mizer" power management for notebook computers using the 8XC752 microcontroller

AN436

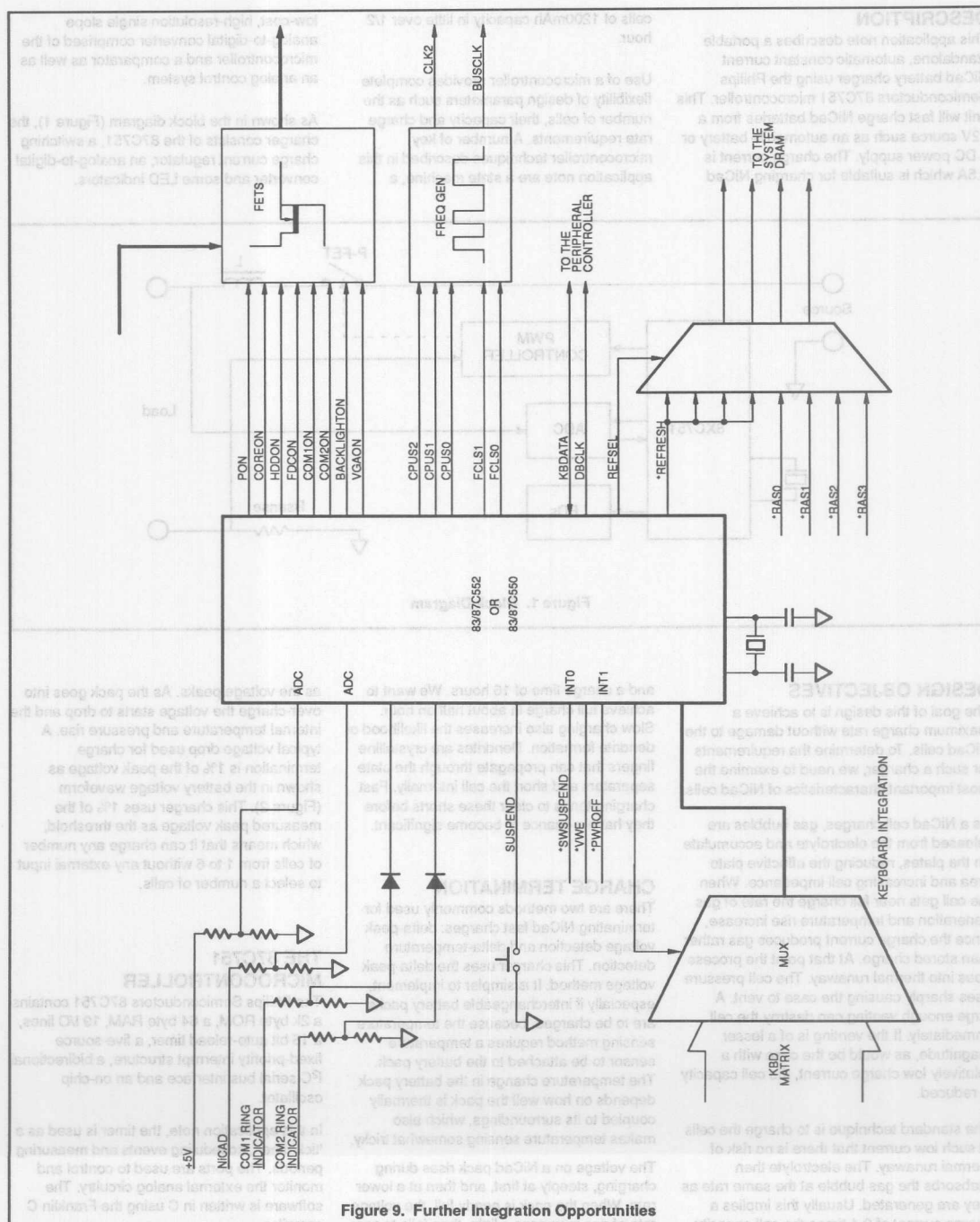


Figure 9. Further Integration Opportunities

87C751 fast NiCad charger

AN439

DESCRIPTION

This application note describes a portable standalone, automatic constant current NiCad battery charger using the Philips Semiconductors 87C751 microcontroller. This unit will fast charge NiCad batteries from a 12V source such as an automobile battery or a DC power supply. The charge current is 2.5A which is suitable for charging NiCad

cells of 1200mAh capacity in little over 1/2 hour.

Use of a microcontroller provides complete flexibility of design parameters such as the number of cells, their capacity and charge rate requirements. A number of key microcontroller techniques described in this application note are a state machine, a

low-cost, high-resolution single slope analog-to-digital converter comprised of the microcontroller and a comparator as well as an analog control system.

As shown in the block diagram (Figure 1), the charger consists of the 87C751, a switching charge current regulator, an analog-to-digital converter and some LED indicators.

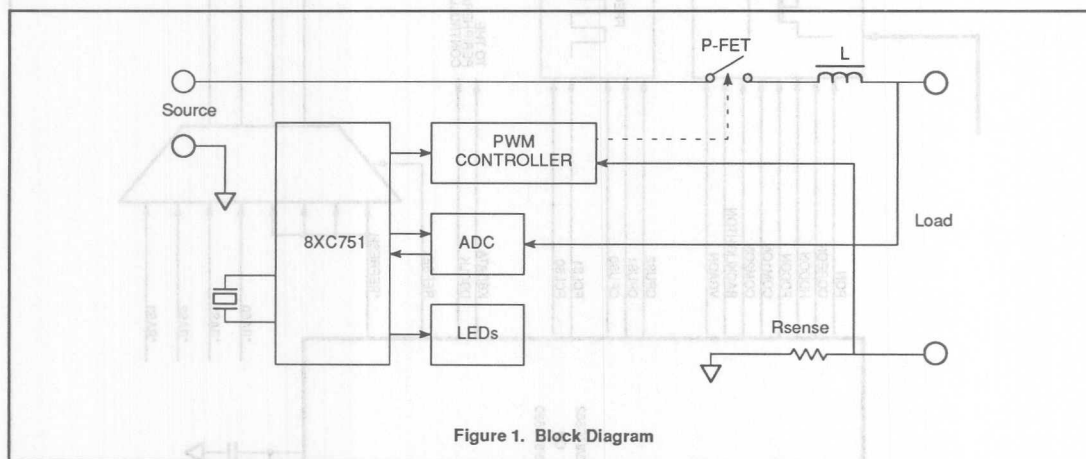


Figure 1. Block Diagram

DESIGN OBJECTIVES

The goal of this design is to achieve a maximum charge rate without damage to the NiCad cells. To determine the requirements for such a charger, we need to examine the most important characteristics of NiCad cells.

As a NiCad cell charges, gas bubbles are released from the electrolyte and accumulate on the plates, reducing the effective plate area and increasing cell impedance. When the cell gets near full charge the rate of gas generation and temperature rise increase, since the charge current produces gas rather than stored charge. At that point the process goes into thermal runaway. The cell pressure rises sharply causing the case to vent. A large enough venting can destroy the cell immediately. If the venting is of a lesser magnitude, as would be the case with a relatively low charge current, the cell capacity is reduced.

The standard technique is to charge the cells at such low current that there is no risk of thermal runaway. The electrolyte then reabsorbs the gas bubble at the same rate as they are generated. Usually this implies a charge current of 0.1 times the cell capacity

and a charge time of 16 hours. We want to achieve full charge in about half an hour. Slow charging also increases the likelihood of dendrite formation. Dendrites are crystalline fingers that can propagate through the plate separators and short the cell internally. Fast charging tends to clear these shorts before they have a chance to become significant.

CHARGE TERMINATION

There are two methods commonly used for terminating NiCad fast charges: delta-peak voltage detection and delta-temperature detection. This charger uses the delta-peak voltage method. It is simpler to implement, especially if interchangeable battery packs are to be charged, because the temperature sensing method requires a temperature sensor to be attached to the battery pack. The temperature change in the battery pack depends on how well the pack is thermally coupled to its surroundings, which also makes temperature sensing somewhat tricky.

The voltage on a NiCad pack rises during charging, steeply at first, and then at a lower rate. When the pack is nearly full, the voltage rate of rise increases a little, then falls to zero

as the voltage peaks. As the pack goes into over-charge the voltage starts to drop and the internal temperature and pressure rise. A typical voltage drop used for charge termination is 1% of the peak voltage as shown in the battery voltage waveform (Figure 2). This charger uses 1% of the measured peak voltage as the threshold, which means that it can charge any number of cells from 1 to 6 without any external input to select a number of cells.

THE 87C751 MICROCONTROLLER

The Philips Semiconductors 87C751 contains a 2k byte ROM, a 64 byte RAM, 19 I/O lines, a 16 bit auto-reload timer, a five-source fixed-priority interrupt structure, a bidirectional I²C serial bus interface and an on-chip oscillator.

In this application note, the timer is used as a 'tick-timer', scheduling events and measuring periods. The ports are used to control and monitor the external analog circuitry. The software is written in C using the Franklin C compiler.

87C751 fast NiCad charger

AN439

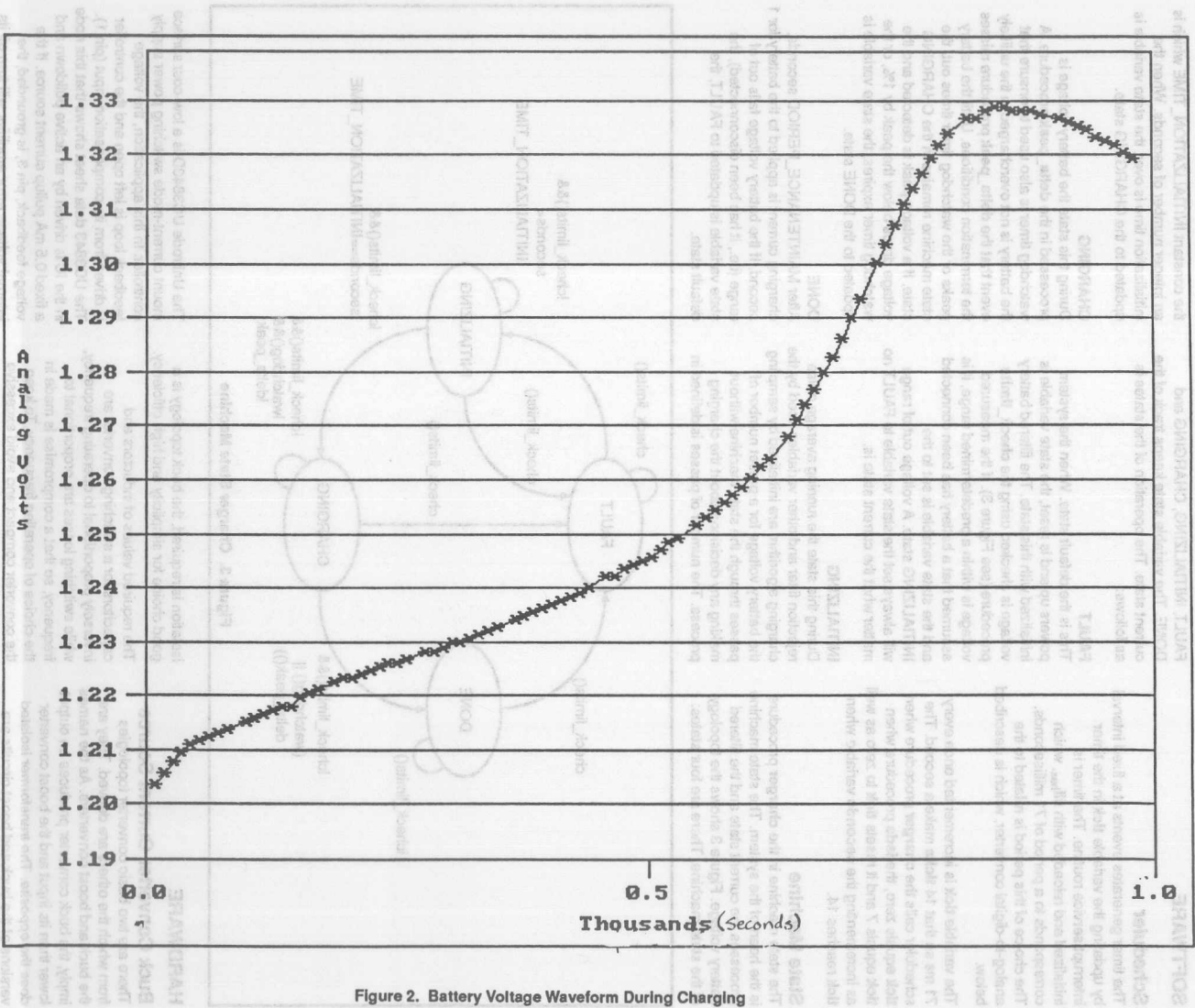


Figure 2. Battery Voltage Waveform During Charging

87C751 fast NiCad charger

AN439

SOFTWARE

Scheduler

The timer generates events at a fixed interval by updating the variable `tick` in the timer interrupt service routine. The timer is initialized and reloaded with `ffffhex`, which corresponds to a period of 71 milliseconds. The choice of this period is related to the analog-to-digital converter, which is described below.

The variable `tick` is incremented once every 71 ms so that 14 ticks make a second. The scheduler calls the `charger` procedure when `tick` equals zero, the `leds` procedure when `tick` equals 7 and it resets `tick` to zero as well as incrementing the `seconds` variable when `tick` reaches 14.

State Machine

The state machine in the `charger` procedure is the heart of the system. The state machine processes the current state and the filtered battery voltage. Figure 3 shows the topology of the state machine. There are four states:

FAULT, INITIALIZING, CHARGING and DONE. The variable state keeps track of the current state. The operation of the states is as follows:

FAULT

This is the default state. When the system powers up and is reset, the state variable is initialized with this state. The filtered battery voltage is checked using the `check_limits` procedure (see Figure 3). If the measured voltage is within a predetermined range, it is assumed that a battery has been connected and the state variable is set to the **INITIALIZING** state. A voltage out of range will always set the state variable to **FAULT** no matter what the current state is.

INITIALIZING

During this state the running average noise rejection filter and other variables used by the charging algorithm are initialized by sampling the battery voltage for a preset number of passes through the state machine without making any decisions about the charging process. The number of passes is defined in

the constant `INITIALIZATION_TIME` which is an integer number of seconds. When the initialization time is over, the state variable is updated to the **CHARGING** state.

CHARGING

During this state the battery voltage is processed in the `delta_peak` procedure. A watchdog timer is also used to ensure that the battery is not overcharged in the unlikely event that the `delta_peak` procedure misses the termination conditions. Until the battery peaks or the watchdog timer times out, the state machine remains in the **CHARGING** state. If a voltage peak is detected and the voltage drops below the peak by 1%, or the watchdog timer expires, the state variable is updated to the **DONE** state.

DONE

After `MAINTENANCE_PERIOD` seconds, charging current is applied to the battery for 1 second. If the battery voltage falls out of range (i.e., it has been disconnected), the state variable is updated to **FAULT**, the default state.

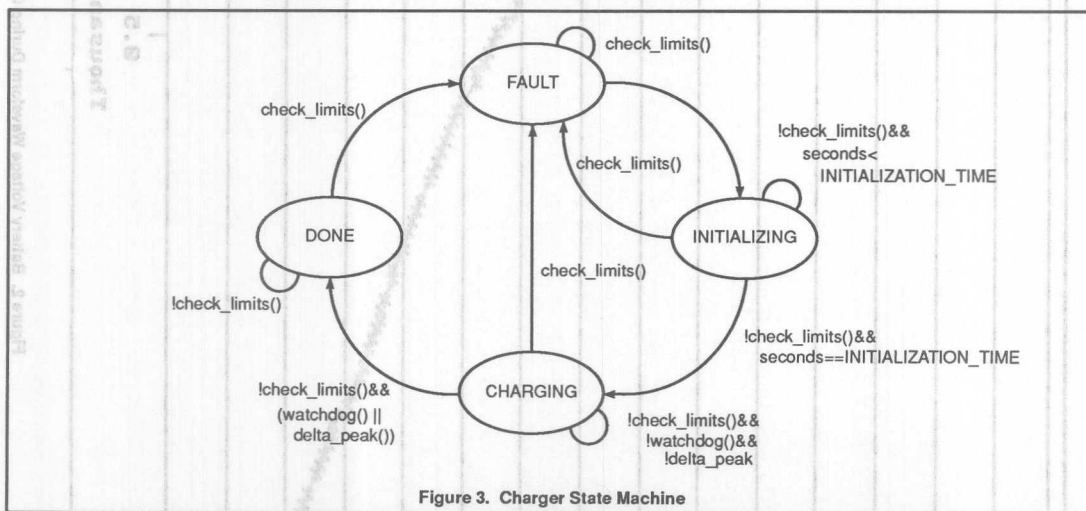


Figure 3. Charger State Machine

HARDWARE

Buck Converter Current Source

There are two basic converter topologies from which the others are derived. They are the buck and boost converters. As the names imply, the buck converter produces an output lower than its input and the boost converter does the opposite. The transformer isolated versions of the buck and boost circuits are known as the forward and flyback converters. Since the input supply voltage is greater than the maximum expected output voltage and no

isolation is required, the buck topology is a good choice for simplicity and high efficiency.

The required values of inductors and capacitors for a switching converter are inversely proportional to operating frequency, while switching losses are proportional to frequency, so that a compromise is made in the choice of operating frequency. To keep the converter compact and avoid excessive switching losses, the switching frequency was chosen at 100kHz.

The Unitrode UC3843D is a low-cost surface mount current-mode switching power supply controller. In this application, the voltage feedback loop is left open and the controller is driven from its compensation input (pin 1). The UC3843 data sheet shows that this node in the IC is driven by an active pulldown and a fixed 0.5 mA pullup current source. If the voltage feedback, pin 3, is grounded, the internal voltage error amplifier will turn off its output, allowing control of the node voltage by pulling current out of pin 1.

87C751 fast NiCad charger

AN439

The ICL7667 is an inverting MOS gate driver. It provides the correct polarity for the mosfet gate signal and its 1.5A peak output improves efficiency by switching the fet quickly. The Amobead is an optional component which reduces EMI at the expense of increased drain voltage swing. The scope traces were taken without the Amobead in place.

Analog-to-Digital Converter

The analog-to-digital converter consists of Q104, C118, R114, U104 as well as R115, R116 and C119 operating in conjunction with the 87C751. In this application a simple, low-cost ADC with relatively high resolution is required, but it does not need linearity, absolute accuracy or long term stability because the detection method is relative to the peak voltage and happens over a period less than one hour. The circuit functions as a voltage-to-period converter. Although the capacitor voltage follows an exponential curve, it is nearly linear in the operating region from 0V to the comparison threshold of 454mV since the battery voltage which drives it is typically 5 to 6V for a four cell pack under charge. With a single cell the period can stretch to near the 71ms tick period. The capacitor voltage is described by the equation

$$V_C = V_{BATT}(1 - e^{-t/RC})$$

where $C=0.1\mu F$ and $R=1M$ in this circuit. The straight line approximation we are using is the tangent to the actual curve at $t=0$. This can be found by differentiating the above equation with respect to time

$$dV_C/dt = V_{BATT}/RC \cdot e^{-t/RC}$$

and setting $t=0$. Then the expression becomes

$$dV_C/dt = V_{BATT}/RC$$

Integrating the above expression yields a straight line

$$V_C = V_{BATT}/RC \cdot t$$

from the origin to the point (RC, V_{BATT}) . If you solve for t using the straight line and $V_C=454mV$, $V_{BATT}=5.50V$, you get $t=8.255ms$. Substituting this back into the first equation and solving for V_{BATT} yields 5.73V, which is within 5% of the actual voltage. The absolute accuracy of the conversion is only important when using the battery voltage measurement to detect a battery connected to the output, versus a short circuit or an open circuit. All the critical sensing is done within 1% of the peak voltage and is relative to the peak.

To avoid contamination of the battery voltage reading by switching noise, the voltage sensing is done during a quite period when the switching current source is turned off by the processor.

Waveforms

The first trace (Figure 4) shows drain to ground voltage on the P-channel FET. Note that the peak negative voltage is -17.66V and there is minimal flyback ringing. The positive spike on turn-on may be due to sense resistor inductance.

The second trace (Figure 5) shows the voltage across the 0.1Ω sense resistor at a DC output current of 2.5A and an input voltage of 12.2V. Voltage spikes due to trace and sense resistor inductance are filtered out by R105, C112 to prevent false triggering of the UC3843 current comparator.

The third trace (Figure 6) shows the FET gate voltage. FET gates are usually rated at ±20V maximum, but reliability is enhanced if they are kept within ±15V. In this case the gate voltage is well within range for reliable operation.

REFERENCES

1. Billings, Keith H.: Switchmode Power Supply Handbook, McGraw Hill 1989
2. Unitrode I.C. Data Handbook, Unitrode Corp. 1990
3. Panasonic NiCad Battery Handbook

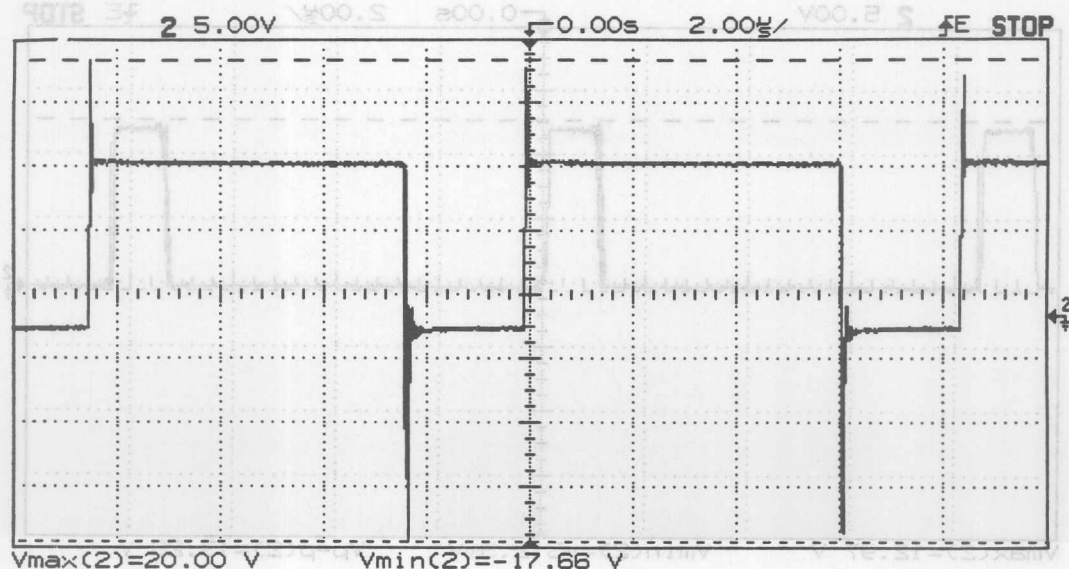


Figure 4. First Trace

87C751 fast NiCad charger

AN439

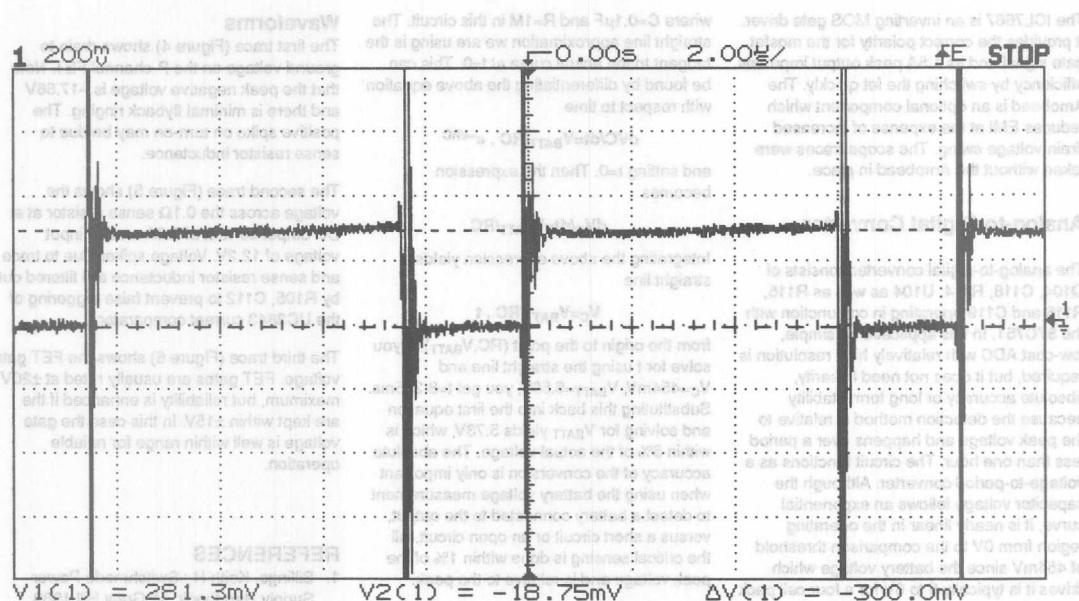


Figure 5. Second Trace

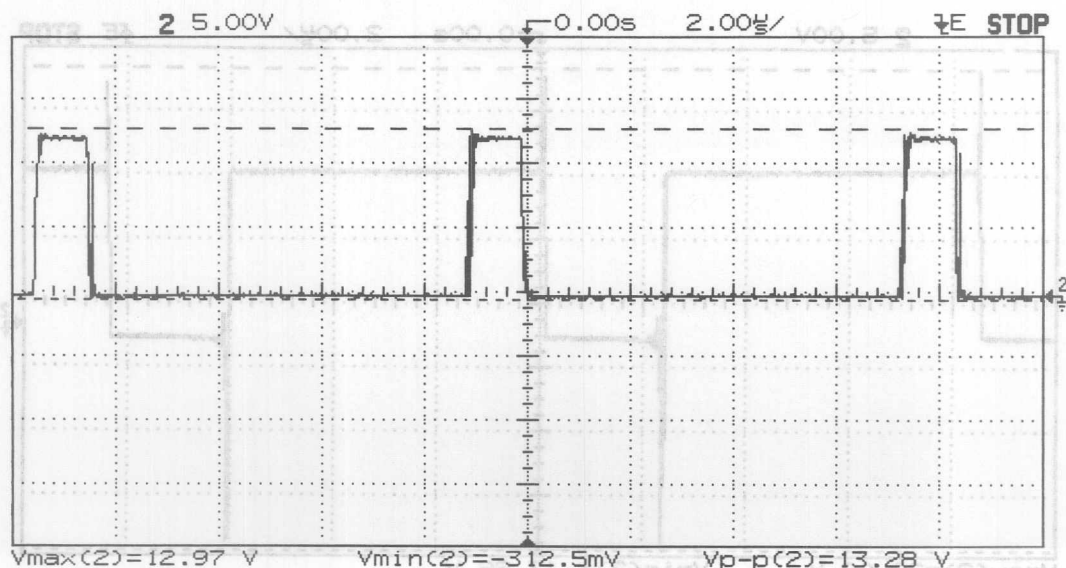
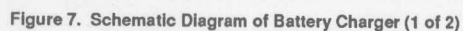


Figure 6. Third Trace

AN439



AN439

Figure 7. Schematic Diagram of Battery Charger (2 of 2)

87C751 fast NiCad charger

87C751 fast NiCad charger AN439

```

#pragma PL (60)
#pragma PW (120)
#pragma OT (3)
#pragma ROM (SMALL)

#include <reg751.h>

typedef unsigned char byte;
typedef unsigned short word;
typedef unsigned long dword;

#define TRUE 1
#define FALSE 0
#define ON 0
#define OFF 1
#define FLASH 2

/* parameters */
#define ONE_OVER_DELTA 128
#define ONE_SECOND 14

/* timers */
/* 1 cpu cycle = 1.085uS */
/* 1 tic = 0xffff cpu cycles */
/* 1 "second" = 14 tics = 995.5uS */
#define MAX_BATT_PERIOD 5530 /* 60ms period, 0.9 volts */
#define MIN_BATT_PERIOD 430 /* 4.67ms period, 10 volts */
#define DEAD_MAN_TIMEOUT 2712 /* 45 minutes */
#define MAINTENANCE_PERIOD 500 /* 1 pulse in 500 Seconds */
#define INITIALIZATION_TIME 20

/* errors */
#define BATTERY_VOLTAGE_OUT_OF_RANGE 0x01
#define BATTERY_CHARGED 0x02
#define DM_TIMEOUT 0x03

/* states */
#define FAULT 0x01
#define INITIALIZING 0x02
#define CHARGING 0x03
#define DONE 0x04

/* global variables */
byte tic;
byte state;
word seconds;
word this_period;
word last1;
word last2;
word last3;
word last4;
word last5;
word last6;
word last7;
word valley;

```

```

sbit comp_out      = 0x90;    /* P1.0 */
sbit clear_cap     = 0x93;    /* P1.3 */
sbit charge        = 0x97;    /* P1.7 */
sbit fault_led     = 0x80;    /* P0.0 */
sbit charge_led    = 0x82;    /* P0.2 */

```

```

word
measure_batt(void) {
    byte tic_now;
    word interval;

    tic_now = tic;

    interval = 0;
    clear_cap = FALSE;
    while(!comp_out && tic==tic_now)
        interval++;
    clear_cap = TRUE;
    return (interval);
}

```

```

byte
check_limits(
    word batt_period
) {
    if ((batt_period > MIN_BATT_PERIOD) && (batt_period < MAX_BATT_PERIOD)) {
        return(FALSE);
    }
    else {
        return(BATTERY_VOLTAGE_OUT_OF_RANGE);
    }
}

```

```

word
filter (
    word last0
) {
    word temp1, temp2, temp3, temp4;
    word result1, result2;

    temp1 = ((last0 / 2) + (last1 / 2));
    temp2 = ((last2 / 2) + (last3 / 2));
    temp3 = ((last4 / 2) + (last5 / 2));
    temp4 = ((last6 / 2) + (last7 / 2));

    result1 = ((temp1 / 2) + (temp2 / 2));
    result2 = ((temp3 / 2) + (temp4 / 2));

    last7 = last6;
    last6 = last5;
    last5 = last4;
    last4 = last3;
    last3 = last2;
    last2 = last1;
    last1 = last0;
}

```

87C751 fast NiCad charger

AN439 fast NiCad charger

```

        return((result1 / 2) + (result2 / 2));
    }

    byte
    delta_peak (
        word    period
    ) {
        if (period < valley)
            valley = period;
        if (period > (valley + (valley/ONE_OVER_DELTA)))
            return (BATTERY_CHARGED);
        else
            return (FALSE);
    }

    byte
    watchdog (
        word    now
    ) {
        if (now < DEAD_MAN_TIMEOUT)
            return (FALSE);
        else
            return (DM_TIMEOUT);
    }

    void
    charger ( void ) {

        this_period = measure_batt();
        this_period = filter(this_period);

        switch (state) {
            case FAULT: {
                if (!check_limits(this_period)) {
                    seconds = 0;
                    state = INITIALIZING;
                }
                else
                    state = FAULT;
                break;
            }
            case INITIALIZING: {
                if (check_limits(this_period))
                    state = FAULT;
                else {
                    charge = TRUE;
                    if (seconds < INITIALIZATION_TIME)
                        state = INITIALIZING;
                    else {
                        valley = 0xffff;
                        state = CHARGING;
                    }
                }
                break;
            }
            case CHARGING: {
                if (check_limits(this_period))
                    state = FAULT;
            }
        }
    }

```

87C751 fast NiCad charger

87C751 fast NiCad charger AN439

```

else {
    if (!watchdog(seconds)) {
        if (delta_peak(this_period)) {
            state = DONE;
            seconds = 0;
        }
        else {
            state = CHARGING;
            charge = TRUE;
        }
    }
    else {
        state = DONE;
        seconds = 0;
    }
}
break;
}
case DONE: {
    if (check_limits(this_period))
        state = FAULT;
    else {
        if (seconds < MAINTENANCE_PERIOD) {
            charge = TRUE;
            seconds = 0;
        }
        state = DONE;
    }
    break;
}
}

void
leds ( void ) {

switch (state) {
case FAULT: {
    charge_led = OFF;
    fault_led = ON;
    break;
}
case INITIALIZING: {
    charge_led = ON;
    fault_led = OFF;
    break;
}
case CHARGING: {
    charge_led = ON;
    fault_led = OFF;
    break;
}
case DONE: {
    charge_led = !charge_led;
    fault_led = OFF;
    break;
}
}
}

```

87C751 fast NiCad charger

AN439

```

/* Timer 0 interrupt */
void
timer0(void) interrupt 1
{
    for further initialization
}

```

```

    tic++;
}

```

```

void
main()
{
    A derivative of the 8051 family of
    microcontrollers, the 87C751 has an 8
    CPU, 2K bytes EPROM, 64 bytes RAM, 18
    NO lines, a bi-directional inter-integrated
    circuit (PIC) serial bus interface, and an
    on-chip timer.
}

```

```

/* initialize pins */
charge=FALSE;
charge_led = OFF;
fault_led = OFF;

```

```

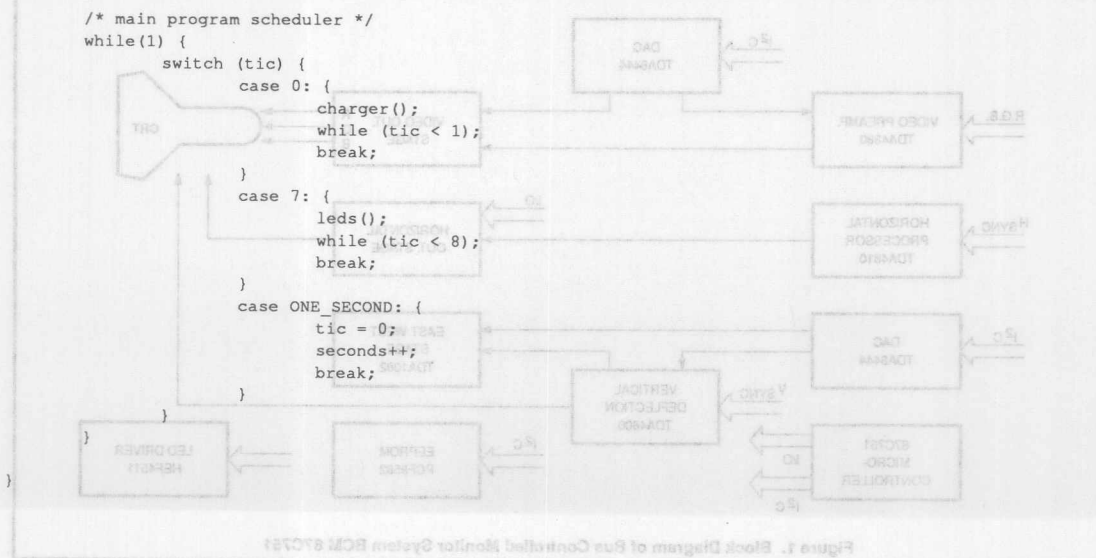
/* initialize timer */
TR1=1;
CT=0;
GATE=0;
RTH=0;
RTL=0;
ITO = TRUE;
ET0=1;
EA=TRUE;

```

```

/* initialize globals */
tic=0;
seconds = 0;
valley = 0xffff;
state = FAULT;

```



The system can perform the following:

- Determining the mode and standard of incoming signals with the stored values in memory (e.g., multisync modes).
- Error parameters of user defined modes into memory via a keyboard.
- Control analog parameters such as contrast and brightness via the bus from keyboard inputs.
- Control mode and standard parameters (such as picture geometry parameters and the free-running oscillator frequency).

Features

- Multisync or Autosync operation
- PH = 31kHz-68kHz
- PV = 50Hz-11kHz
- Selectable four or five function control configurations
- Selectable one digit, or one and one-half digit, or full seven segment mode display.
- Selectable Horizontal direct ratio or indirect ratio P-V converter DAC output configurations
- Selectable ten user modes plus ten factory modes, or one user mode plus 19 factory mode configurations
- Selectable multiple keyboard keys or minimum keyboard configuration
- Change function without save key, all keys but the release key have a repeat function
- Both Horizontal and Vertical outputs have P-to-V converter DAC outputs

SUMMARY

BCM87C751 is a powerful, flexible and low cost Digital Control Monitor System, based on the 87C751 microcontroller. It employs P-C bus control with various P-C bus controlled peripherals (P-CB8252EP, EEPROM and TD8444M DAC converter). The control function is implemented via 8-bit DC voltage output from TD8444M.

Some features of the system:

- Flexible approach, especially for multisync or auto sync operation
- Mode detection and redundancy measurements by microprocessor
- Mode switching under software control
- Elimination of potentiometers
- Quick factory alignment (DACS can be pressed)
- Automatic factory alignment possible

This document describes the operation and the use of the system. It provides necessary information concerning operation, required hardware, flow charts and their effect on the performance.

INTRODUCTION

Figure 1 shows the block diagram of a high-performance color monitor with microcontroller and several parts that communicate via the two-wire P-C-bus.

(BCM) 87C751

Specification for a bus-controlled monitor

AN442

SUMMARY

BCM87C751 is a powerful, flexible and low cost Digital Controlled Monitor System, based on the 87C751 microcontroller. It employs I²C bus control with various I²C bus controlled peripherals (PCF8582EP-EEPROM and TDA8444 D/A converter). The control function is implemented via 8 6-bit DC voltage output from TDA8444.

Some features of the system:

- Flexible approach, especially for multisync or auto sync operation
- Mode detection and frequency measurements by microprocessor
- Mode switching under software control
- Elimination of potentiometers
- Quick factory alignment (DACs can be preset)
- Automatic factory alignment possible

This document describes the operation and the use of the system. It provides necessary information concerning operation, required hardware, flow charts and their effect on the performance.

INTRODUCTION

Figure 1 shows the block diagram of a high-performance color monitor with microcontroller and several parts that communicate via the two-wire I²C-bus.

The system can perform the following:

- Determine the mode and standard of incoming signals with the stored values in memory (e.g., multisync modes).
- Enter parameters of user defined modes into memory via a keyboard.
- Control analog parameters such as contrast and brightness via the bus from keyboard inputs.
- Control mode and standard parameters (such as picture geometry parameters and the free-running oscillator frequency).

Features

- Multisync or Autosync operation
 - FH = 31kHz–95kHz
 - FV = 50Hz–114Hz
- Selectable four or five function control configurations
- Selectable one digit, or one and one-half digit, or null seven segment mode display.
- Selectable Horizontal direct ratio or indirect ratio F to V converter DAC output configurations
- Selectable ten user modes plus ten factory modes, or, one user mode plus 19 factory modes configurations
- Selectable multiple up/down keys or minimum keyboard configuration
- Change function without save key, all keys but the reload key have a repeat function
- Both Horizontal and Vertical outputs have F-to-V converter DAC outputs

- Four outputs of Horizontal PLL capacitor selection signal. This can be easily adapted for further extension.

IC DESCRIPTIONS

87C751

A derivative of the 8051 family of microcontrollers, the 87C751 has an 8-bit CPU, 2k bytes EPROM, 64 bytes RAM, 19 I/O lines, a bi-directional inter-integrated circuit (I²C) serial bus interface, and an on-chip oscillator.

PCF8581/2

A 1k- or 2k-bit, 5V electrically erasable programmable read only memory (EEPROM) organized as 128 or 256 × 8 bits. The stored information is electrically alterable on a word-by-word basis, I²C-bus controlled.

TDA8444

Comprises eight DACs, each controlled via the I²C-bus. The DACs are individually programmed using a 6-bit word to select an output from one of 64 voltage steps. the maximum output voltage of all DACs is set by the input VMAX and the resolution is approximately VMAX/64.

For detailed information on these devices, please refer to their respective data sheets.

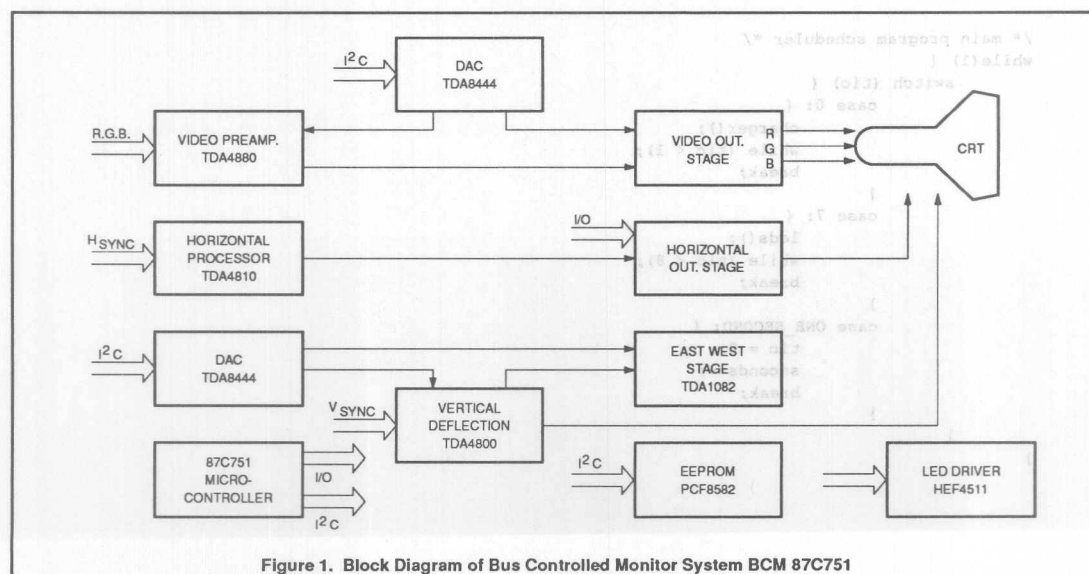


Figure 1. Block Diagram of Bus Controlled Monitor System BCM 87C751

(BCM) 87C751

Specification for a bus-controlled monitor

AN442

OPERATION INSTRUCTION

Function Selection and Change

When the monitor is powered on it will automatically enter the corresponding mode depending on the input signal. Different configurations and functions can be defined by users with the use of different features. The following configuration's functions are examples for user's reference. See Figure 2 for Configurations 1 through 4.

To adjust functions such as V-size, V-shift, H-center, H-shift, and PCC (Pincushion Correction Circuitry):

1. The "Function" key should be pressed until the required function LED is lit.
2. If the V-shift LED is on, the user can then adjust V-shift by pressing Up or Down key. If the Up key is pressed, the V-shift DAC output will increase one step. While the Down key is pressed, the V-shift DAC output will decrease one step. The user can repeat the Up or Down key simply by pressing it longer than 0.5 second. It will then automatically repeat approximately 2 times per second until the key is released.

Mode Selection and Change

(See Figure 4)

In Figures 3 and 4, the mode number is displayed on the two seven-segment LEDs.

Each number denotes one mode. Modes 10 through 19 are user modes, which can be defined by the user. When there is a new mode entering the monitor that does not belong to any mode stored in the EEPROM, the mode display will show "19". If the user presses the Reload key while the mode display is "19", the display will flash. When the mode display is flashing, the user can select the destination mode by pressing the Up/Down keys. The destination mode is between 10 and 19.

Every press of the Up key causes the flashing display to add one, unless it already reached 19. Every press of the Down key causes the flashing display to subtract one, unless it has reached 10. When the user lets the destination mode flash on the display, the user can press the Reload key to store the new mode to destination mode. When the mode display stops flashing, the new mode is stored. The newly stored destination mode is permanent, unless the user repeats the entire procedure.

To change an old user mode, already stored in EEPROM, to a different user mode, press the Reload key for longer than 8 seconds while the monitor is working in the old mode. The mode display will flash the old mode, then the user can use the Up/Down key to select the new mode. Press the Reload key again to copy the old to new destination user

mode. If the user forgets to press the Reload key again, the flashing of the mode display will last for 2 minutes, then the program will cancel the copy old mode to new user mode command.

During the flash period, the program still monitors the Horizontal and Vertical sync signal to adapt the DAC to the proper mode. For example, while the user tries to copy new mode 13 to mode 15, and the mode display 13 (15) is flashing, if the PC sends out a signal for mode 3, the program will change the DAC output to adjust the monitor to work in mode 3, but the mode display is still flashing on user mode 13 (15) and the store procedure is still going on until the user presses the Reload key again, or terminates the copy procedure after the 2 minutes time out. The mode display then shows 3.

NOTE: Upon request, we can also program in advance those 10 user-defined modes that can still be changed by the user if necessary for further extension.

For Configuration 5 to Configuration 8, see Figure 3.

To adjust functions, the user can simply press the corresponding push button. The upper push button will increase the DAC output. The lower push button will decrease the DAC output. (A total of 64 steps can be programmed in advance.)

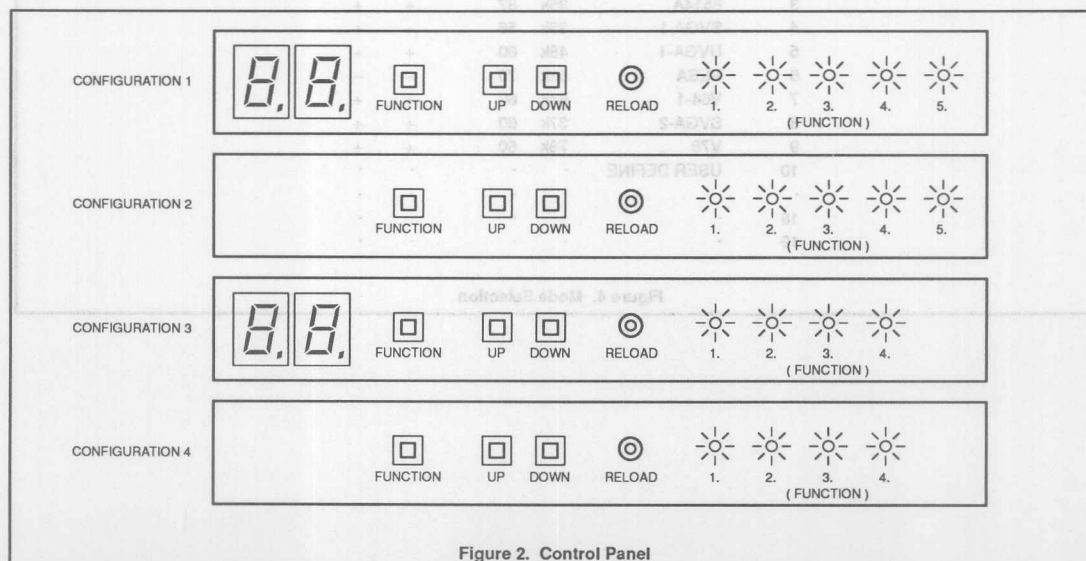


Figure 2. Control Panel

(BCM) 87C751
Specification for a bus-controlled monitor

AN442

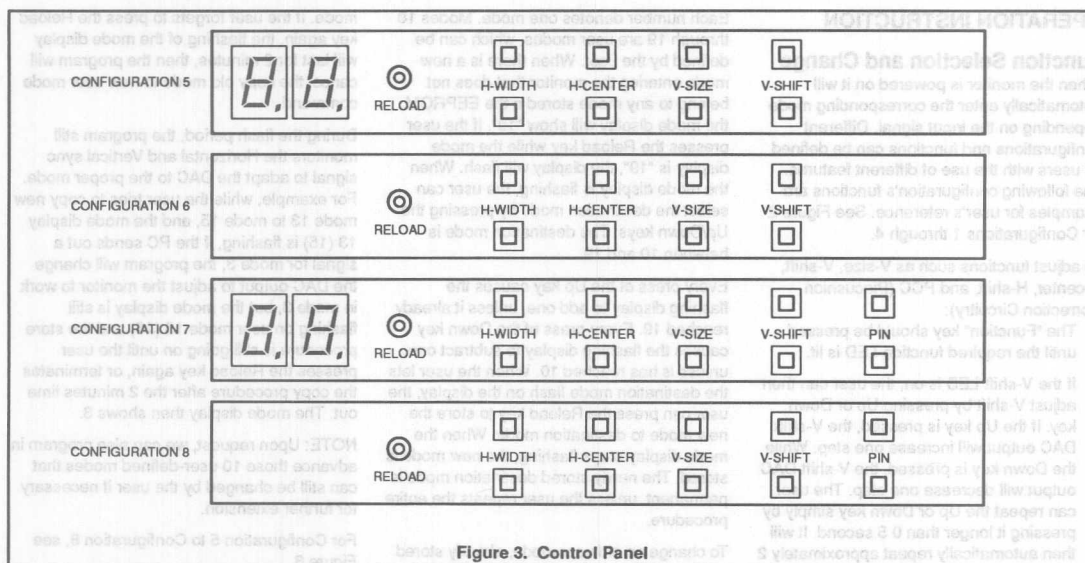


Figure 3. Control Panel

MODE	NAME	H.F.	V.F.	H.P.	V.P.
0	VGA-1	31k	70	-	-
1	VGA-2	31k	70	-	+
2	VGA-3	31k	60	-	-
3	8514A	35k	87	+	+
4	SVGA-1	35k	56	+	+
5	UVGA-1	48k	60	+	+
6	VESA	56k	70	-	-
7	V64-1	64k	60	+	+
8	SVGA-2	37k	60	+	+
9	V78	78k	60	+	+
10	USER DEFINE
18
19

Figure 4. Mode Selection

Figure 4. Mode Selection

(BCM) 87C751

AN442

Specification for a bus-controlled monitor

SOFTWARE FLOW CHART
DESCRIPTION

(See Figures 5 through 7)

When power is on, software initializes the hardware first. The microcontroller waits 100 μ s for the settlement of the hardware, then initializes itself by specifying stack, setting timer, clearing RAM, arranging interrupt, . . . , etc.

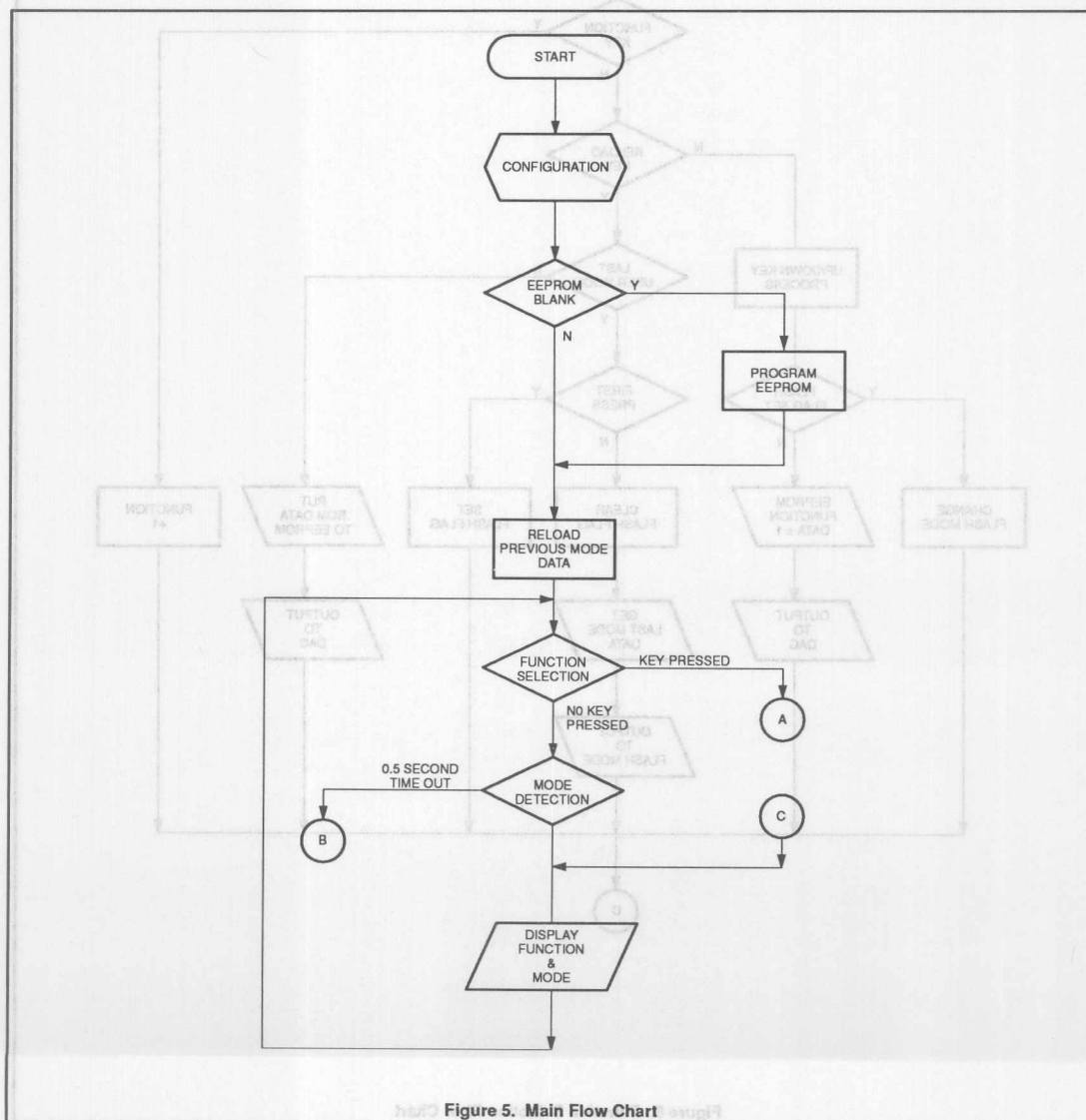
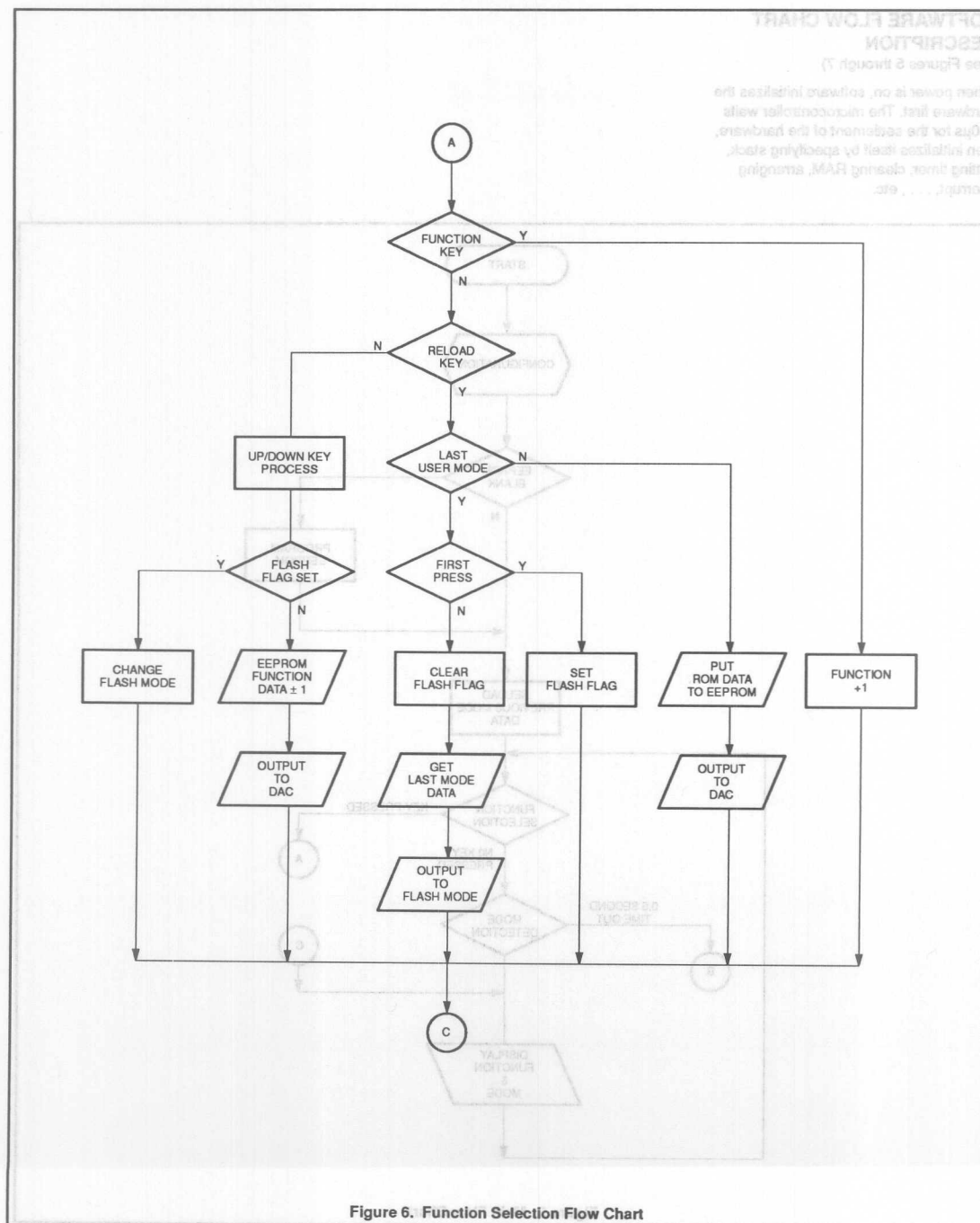
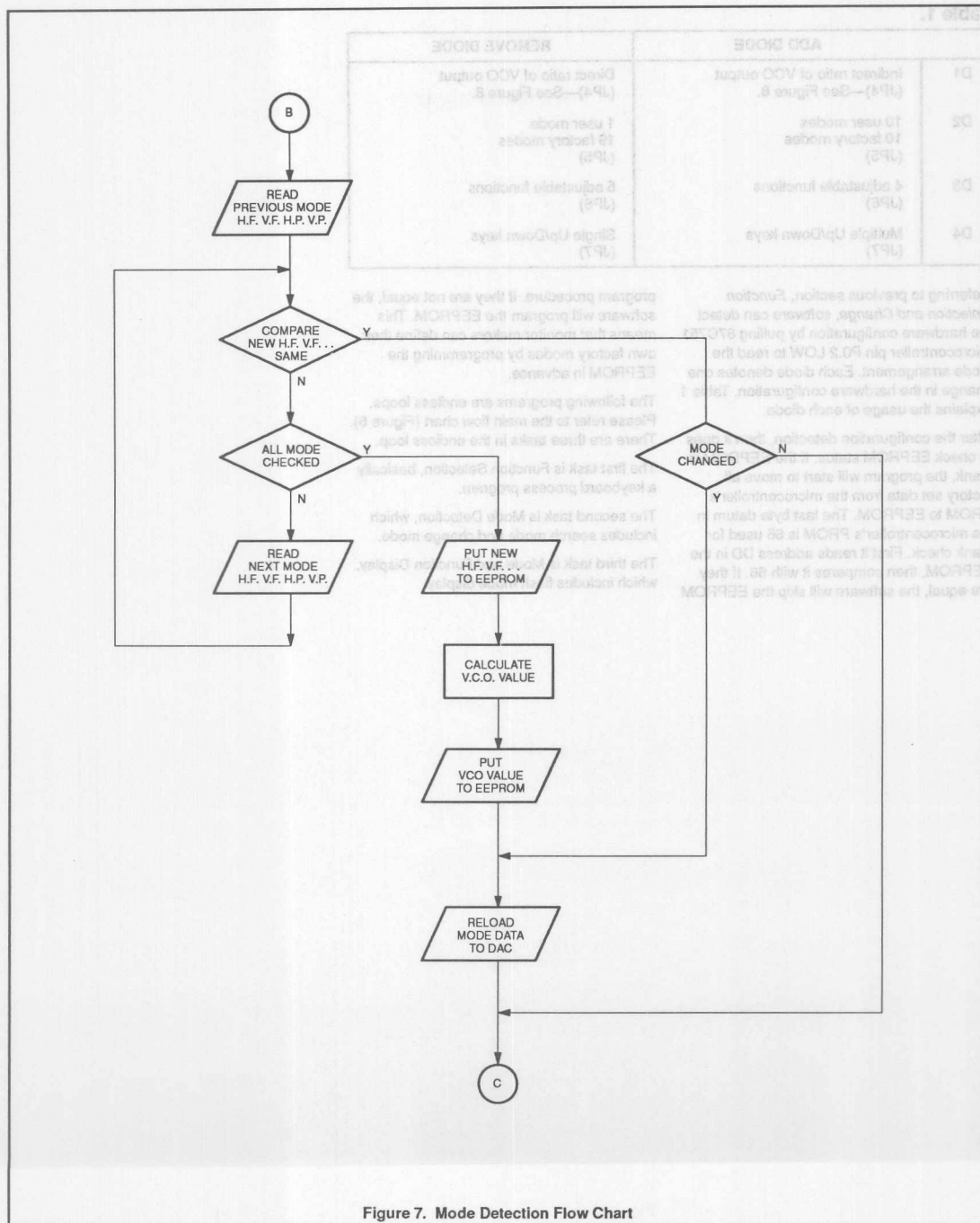


Figure 5. Main Flow Chart

(BCM) 87C751 Specification for a bus-controlled monitor

AN442





(BCM) 87C751

Specification for a bus-controlled monitor

AN442

Table 1.

ADD DIODE		REMOVE DIODE
D1	Indirect ratio of VCO output (JP4)—See Figure 8.	Direct ratio of VCO output (JP4)—See Figure 8.
D2	10 user modes 10 factory modes (JP5)	1 user mode 19 factory modes (JP5)
D3	4 adjustable functions (JP6)	5 adjustable functions (JP6)
D4	Multiple Up/Down keys (JP7)	Single Up/Down keys (JP7)

Referring to previous section, *Function Selection and Change*, software can detect the hardware configuration by pulling 87C751 microcontroller pin P0.2 LOW to read the diode arrangement. Each diode denotes one change in the hardware configuration. Table 1 explains the usage of each diode.

After the configuration detection, then it goes to check EEPROM status. If the EEPROM is blank, the program will start to move all factory set data from the microcontroller's PROM to EEPROM. The last byte datum in the microcontroller's PROM is 66 used for blank check. First it reads address DD in the EEPROM, then compares it with 66. If they are equal, the software will skip the EEPROM

program procedure. If they are not equal, the software will program the EEPROM. This means that monitor makers can define their own factory modes by programming the EEPROM in advance.

The following programs are endless loops. Please refer to the main flow chart (Figure 5). There are three tasks in the endless loop.

The first task is Function Selection, basically a keyboard process program.

The second task is Mode Detection, which includes search mode and change mode.

The third task is Mode and Function Display, which includes flash mode display.

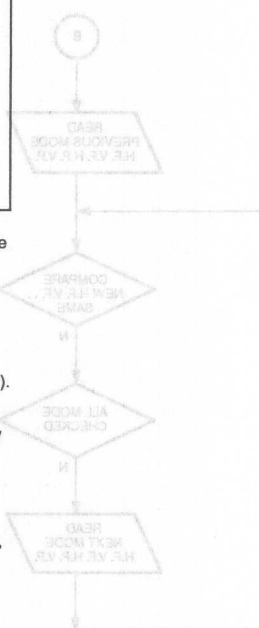


Figure 7. Mode Detection Flow Chart

(BCM) 87C751

Specification for a bus-controlled monitor

AN442

Mode Detection

Beginning with branch "B", Mode Detection Flow Chart (Figure 7), the block at the top of the flow chart is "Read Previous Mode" (the time before 0.5 second ago) and includes Horizontal Sync Frequency, Vertical Sync Frequency, Horizontal Sync Polarity, and Vertical Sync Polarity. The second block is a comparison test block. When current mode (from 0.5 second ago until now) parameters are the same as those in previous mode, the program will branch to the right test block. Since the mode is not changed, the second test block in the right part of the flow chart will branch to leave the Mode Detection section.

If the current mode (from 0.5 second ago until now) parameters are not the same as the previous mode (the time before 0.5 second ago), the first test block from the top of the flow chart will branch to search all mode parameters in the EEPROM to find out what the current mode should be. The left loop of the flow chart checks for the end of the search procedure, i.e., if all modes in the EEPROM are searched and checked, and

the outcome is the same, then this test block will branch to set up a new user mode (19), as per the 4 steps indicated in the central flow chart line.

The first step in setting up a new user mode is to "Put New Parameters" (such as Horizontal Sync Frequency, Vertical Sync Frequency, Horizontal Sync Polarity, and Vertical Sync Polarity) into the EEPROM. The new mode parameters are always saved in the last mode address. If the configuration allowing 10 user modes is selected, then diode 2 is added. If one was found to be the same, the program will branch to the right test block. If it then finds that there is a mode change, it will branch to Reload Mode Data to DAC to complete the mode change procedure.

When the mode change procedure is completed, the monitor will be working in a new mode. Since the program enters the Mode Detection task every 0.5 second, it takes from 0.5 to 1.0 second to finish the change of mode. To save the new user mode (mode 19) to other user mode (10-18), the

user can use the RELOAD key to save the new user mode to other user mode (modes 10 through 18).

If the configuration for 10 user modes is selected, it is highly recommended that you save new user mode to other user mode (10-18) because the last mode "mode 19", will be overwritten by any new user mode whenever a new user mode is detected, after new user mode parameters are detected.

The second step in setting up a new user mode is "Calculating VCO Output Value" (see Figure 8). There are two different curves for the designer to select. If diode 1 is removed, the VCO Output Voltage will be in direct ratio to the Horizontal Sync Frequency. If diode 1 is added, the VCO Output Voltage will have an indirect ratio.

The third step in setting up a new user mode is "Put VCO Value to EEPROM".

The last step is "Reload Mode Data to DAC". After reloading the DAC, the monitor is changed to the new user mode.

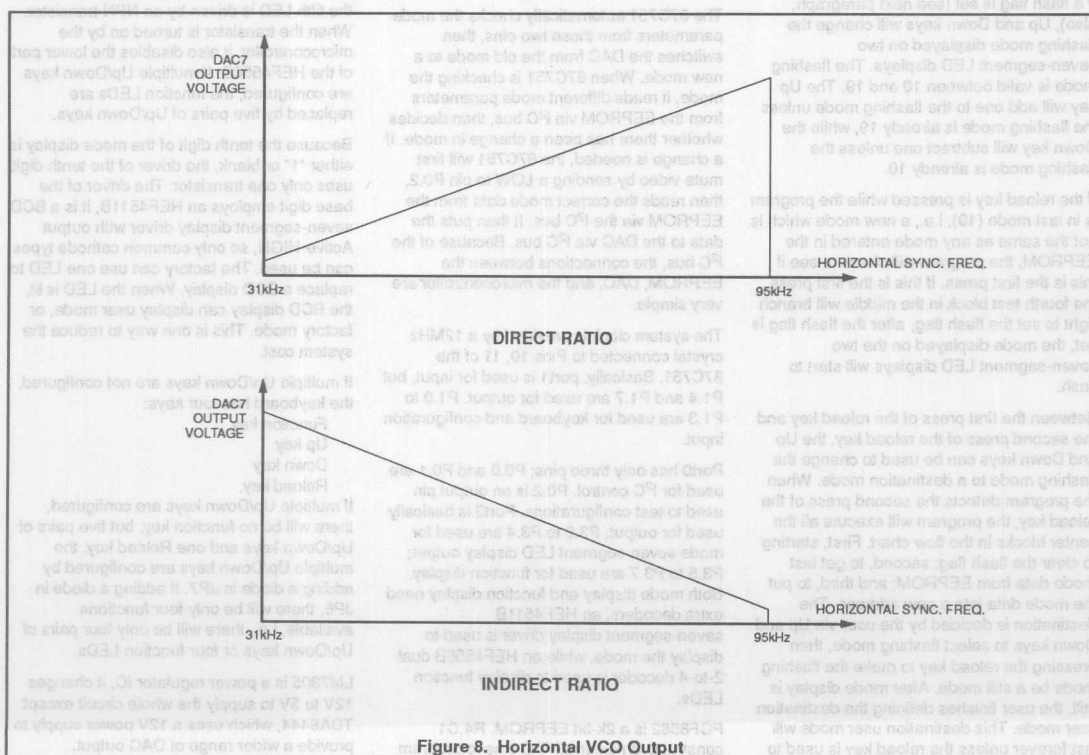


Figure 8. Horizontal VCO Output

Key Function Selection

The first task is Function Selection (see Figure 6). When the program scans the keyboard, it first checks for the function key. If it is depressed, the program will branch right to change function VALUE(+1). the function LEDs are lit in sequence, i.e., when the second LED is lighting and the program detects a press in function key, the program will light the third LED and turn off the second LED. If the program detects that the last LED is lighting, it will turn on the first LED and turn off the last LED.

If the key pressed by the user is not a function key, the program will check if it is the reload key. If it is not, then the second test block will branch left to check Up/Down keys. both the Up and Down keys have two different definitions. When the user is updating function contents, Up and Down keys are used to change the function contents stored in the EEPROM. In that situation, the flash flag is not set; after the program branch left from the reload key test block, the program will test flash flag, then the program will change the output of DAC.

If a flash flag is set (see next paragraph, also), Up and Down keys will change the flashing mode displayed on two seven-segment LED displays. The flashing mode is valid between 10 and 19. The Up key will add one to the flashing mode unless the flashing mode is already 19, while the Down key will subtract one unless the flashing mode is already 10.

If the reload key is pressed while the program is in last mode (19), i.e., a new mode which is not the same as any mode entered in the EEPROM, the program will check to see if this is the first press. If this is the first press, the fourth test block in the middle will branch right to set the flash flag, after the flash flag is set, the mode displayed on the two seven-segment LED displays will start to flash.

Between the first press of the reload key and the second press of the reload key, the Up and Down keys can be used to change the flashing mode to a destination mode. When the program detects the second press of the reload key, the program will execute all the center blocks in the flow chart. First, starting to clear the flash flag; second, to get last mode data from EEPROM; and third, to put the mode data into a new address. The destination is decided by the user via Up and Down keys to select flashing mode, then pressing the reload key to make the flashing mode be a still mode. After mode display is still, the user finishes defining the destination user mode. This destination user mode will last forever unless the reload key is used to redefine it.

If the user presses the reload key, and the program is not in the last mode, it will branch right to reset mode data in EEPROM, starts to read original mode data in microcontroller's PROM, then puts these data to EEPROM, the outputs to DAC. This function is to help the user to restore the monitor when the monitor display is out of control. For example, if the user adjusts the Horizontal phase too broad, then monitor may become out of sync. As a consequence the screen will be a mess, and it is not easy for the user to re-adjust for correction. This feature will minimize possible complaints from customers.

CIRCUIT DESCRIPTION

U1-87C751 is an 8-bit microcontroller and the heart of the Bus-Controlled Monitor. The 87C751 receives Vertical Sync and Horizontal Sync signals from pins P1.5 and P1.6. The R3,C5 in pin P1.5 is a low pass protection circuit. It can prevent the Vertical Sync signals, including Horizontal Sync Pulse, from interfering with the counting of the Vertical Sync Frequency. The R2 in pin P1.6 is only used for protection of 87C751.

The 87C751 automatically checks the mode parameters from these two pins, then switches the DAC from the old mode to a new mode. When 87C751 is checking the mode, it reads different mode parameters from the EEPROM via I²C bus, then decides whether there has been a change in mode. If a change is needed, the 87C751 will first mute video by sending a LOW to pin P0.2, then reads the correct mode data from the EEPROM via the I²C bus. It then puts the data to the DAC via I²C bus. Because of the I²C bus, the connections between the EEPROM, DAC, and the microcontroller are very simple.

The system clock is provided by a 12MHz crystal connected to Pins 10, 11 of the 87C751. Basically, port1 is used for input, but P1.4 and P1.7 are used for output. P1.0 to P1.3 are used for keyboard and configuration input.

Port0 has only three pins; P0.0 and P0.1 are used for I²C control. P0.2 is an output pin used to test configurations. Port3 is basically used for output; P3.0 to P3.4 are used for mode seven-segment LED display output; P3.5 to P3.7 are used for function display. Both mode display and function display need extra decoders, an HEF4511B seven-segment display driver is used to display the mode, while an HEF4556B dual 2-to-4 decoder is used to display function LEDs.

PCF8582 is a 2k-bit EEPROM. R4,C1 constructs an external R-C time to program the EEPROM. In normal operation the

EEPROM needs 30ms to program one byte. C2 is a decoupling capacitor to stabilize the DC supply voltage for PCF8582.

TDA8444 is an octal 6-bit DAC. R7,VR5,C# constructs a reference voltage to define the DAC's maximum output voltage. In practice, the reference voltage must be below 10.5 volts, so R7 is added to prevent the reference voltage from exceeding that limit. C4 is also a decoupling capacitor to stabilize the DC supply voltage for TDA8444.

The upper half of the HEF4556 is used to provide a switch signal to select Horizontal OSC time constant capacitors. The four outputs are Active-LOW. When the Horizontal Sync Frequency (H.S.F.) falls in one of the four ranges, the corresponding output pin will go low. The four ranges are:

(H.S.F. < 35kHz),
(35kHz < H.S.F. < 40kHz),
(40kHz < H.S.F. < 50kHz),
(H.S.F. > 50kHz).

The enable input in the upper part of the HEF4556B can be used to extend the upper limit of the switch signal. The lower part of the HEF4556B is used to display function LEDs, the fifth LED is driven by an NPN transistor. When the transistor is turned on by the microcontroller, it also disables the lower part of the HEF4556B. If multiple Up/Down keys are configured, the function LEDs are replaced by five pairs of Up/Down keys.

Because the tenth digit of the mode display is either "1" or blank, the driver of the tenth digit uses only one transistor. The driver of the base digit employs an HEF4511B, it is a BCD seven-segment display driver with output Active-HIGH, so only common cathode types can be used. The factory can use one LED to replace a BCD display. When the LED is lit, the BCD display can display user mode, or factory mode. This is one way to reduce the system cost.

If multiple Up/Down keys are not configured, the keyboard has four keys:

Function key,
Up key
Down key
Reload key.

If multiple Up/Down keys are configured, there will be no function key, but five pairs of Up/Down keys and one Reload key. the multiple Up/Down keys are configured by adding a diode in JP7. If adding a diode in JP6, there will be only four functions available, i.e., there will be only four pairs of Up/Down keys or four function LEDs.

LM7805 is a power regulator IC, it changes 12V to 5V to supply the whole circuit except TDA8444, which uses a 12V power supply to provide a wider range of DAC output.

(BCM) 87C751

AN442

Specification for a bus-controlled monitor

There is a table to explain the usage of each pin in the JP1 socket. JP2 and JP8 are connected together, JP3 is only used for future automatic alignment (including production line) if necessary. JP4 to JP7 are used to select hardware configurations as previously mentioned.

SPECIFICATION OF THE SYSTEM

1. The input signals to the system are Horizontal Sync and Vertical Sync. The system accepts standard TTL level signals, i.e., $V_{IH} > 2V$, and $V_{IL} < 0.4V$. Horizontal Sync tolerance is $\pm 0.5kHz$, and Vertical Sync tolerance is $\pm 2/-2$ Hz.

2. There are eight DAC output signals. Their maximum output voltage can be preset by setting a voltage on the TDA8444's V_{MAX} pin. The voltage on the V_{MAX} pin must be below 10.5V and also below the voltage on the TDA8444's V_P pin. For other detailed output current characteristics, please refer to Philips data books IC02a, IC02b and 80C51 and Derivative Microcontrollers.

3. The Horizontal switch outputs have four pins. they are standard CMOS B-type buffered outputs. They are Active-LOW, i.e., there will be only one output active at any time. If the designers wants to add ranges in higher Horizontal Sync Frequency, the designer can put extra

circuits onto the demo board. For example, an OPA can be added as a comparator to detect the VCO output. If the VCO output is higher than a certain voltage (VCO's 60kHz output voltage), the OPA will be triggered and the upper half of the HEF 4556 can be disabled by the OPA via HEF4556's Pin 1, when HEF4556 is disabled, the four Horizontal switch outputs will remain HIGH, then the OPA's output can be used as another switch output.

4. Total current consumption is around 25-90mA, depending on the number of LED and seven-segment displays being lit.

PARTS LIST

87C751 BUS CONTROLLED MONITOR

TH-9102/4

Bill of Materials

November 7, 1991

Revised: November 7, 1991

Revision:

12:08:46

Page 1

ITEM	QUANTITY	REFERENCE	PART
1	1	C1	2700pF
2	6	C2, C3, C4, C8, C12, C13	0.1 μ F
3	1	C5	0.01 μ F
4	1	C6	100 μ F
5	2	C7, C11	1 μ F
6	2	C9, C10	33pF
7	5	D1, D2, D3, D4, D5	LED
8	3	JP1, JP2, JP8	Header 16
9	1	JP3	Header 4
10	4	JP4, JP5, JP6, JP7	Jumper (add diode)
11	2	Q1, Q2	BC548
12	1	R1	470R*7
13	11	R2, R3, R8, R9, R10, R11, R14, R15, R17, R19, R20	470R
14	6	R4, R6, R12, R13, R16, R18	22k
15	1	R5	VR10k
16	1	R7	2k
17	2	R21, R22	56R
18	5	S1, S2, S3, S4, S5	SW Pushbutton
19	1	U1	87C751
20	1	U2	HEF4556B
21	1	U3	PCF8582
22	1	U4	TDA8444
23	1	U5	HEF4511B
24	1	U6	LM7805
25	2	U7, U8	DISP-7
26	1	Y1	12MHz

(BCM) 87C751 Specification for a bus-controlled monitor

AN442

PARTS LIST

87C751 BUS CONTROLLED MONITOR
TH-9102/5

Bill of Materials

November 13, 1991

Revised: November 13, 1991

Revision:

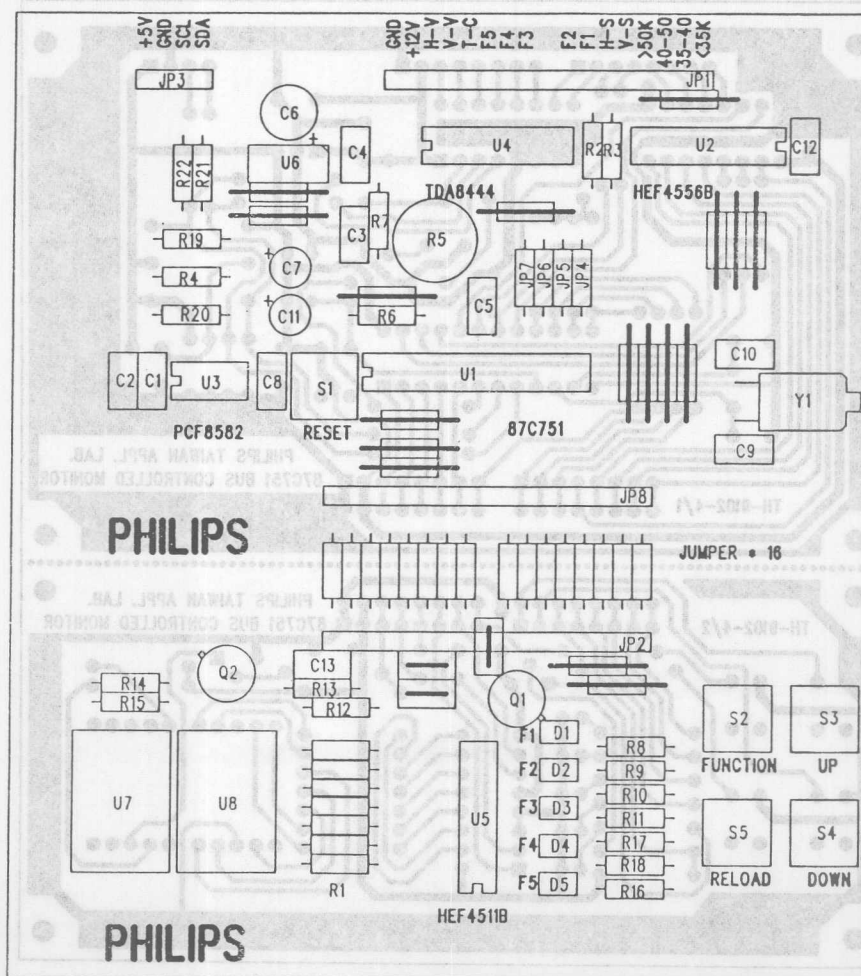
15:42:31 Page 1

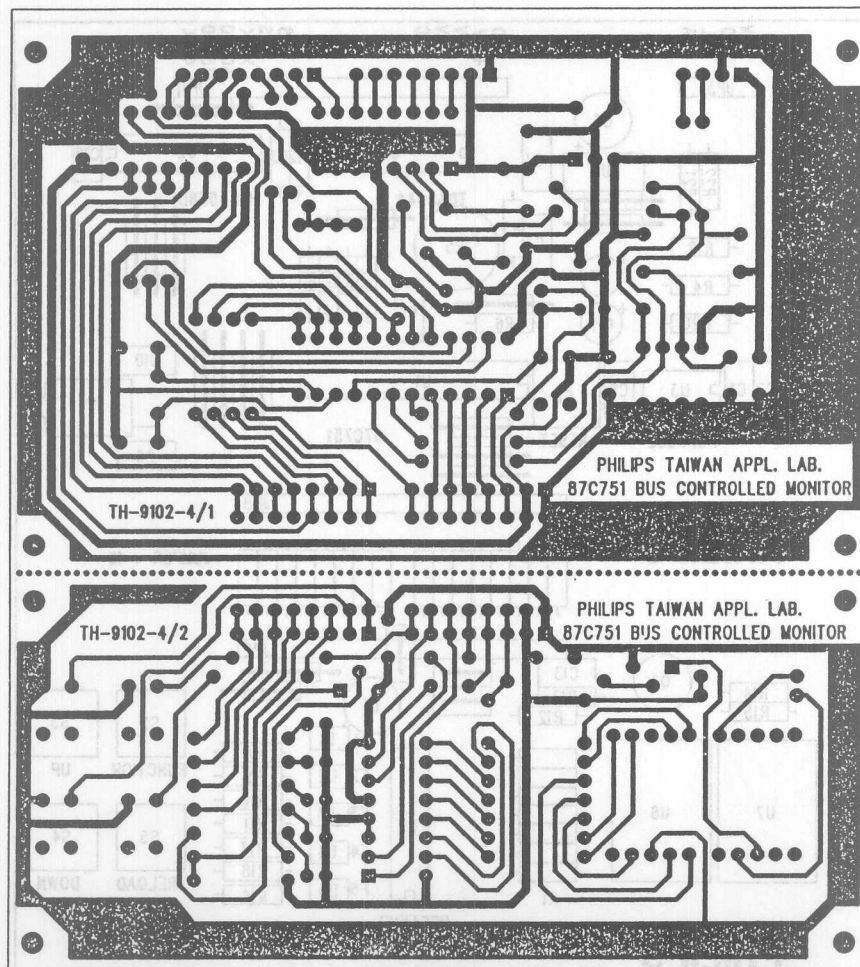
ITEM	QUANTITY	REFERENCE	PART
1	1	C1	2700pF
2	6	C2, C3, C4, C8, C12, C13	0.1µF
3	1	C5	0.01µF
4	1	C6	100µF
5	2	C7, C11	1µF
6	2	C9, C10	33pF
7	3	JP1, JP2, JP8	Header 16
8	1	JP3	Header 4
9	4	JP4, JP5, JP6, JP7	Jumper (add diode)
10	2	Q1, Q2	BC548
11	1	R1	470R*7
12	10	R2, R3, R8, R9, R10, R11, R14, R15, R19, R20	470R
13	6	R4, R6, R12, R13, R16, R18	22k
14	1	R5	VR10k
15	1	R7	2k
16	2	R21, R22	56R
17	12	S1, S2, S3, S4, S5, S6, S7, S8, S9, S10, S11, S12	SW Pushbutton
18	1	U1	87C751
19	1	U2	HEF4556B
20	1	U3	PCF8582
21	1	U4	TDA8444
22	1	U5	HEF4511B
23	1	U6	LM7805
24	2	U7, U8	DISP-7
25	1	Y1	12MHz

(BCM) 87C751 Specification for a bus-controlled monitor

AN442

CIRCUIT DIAGRAM





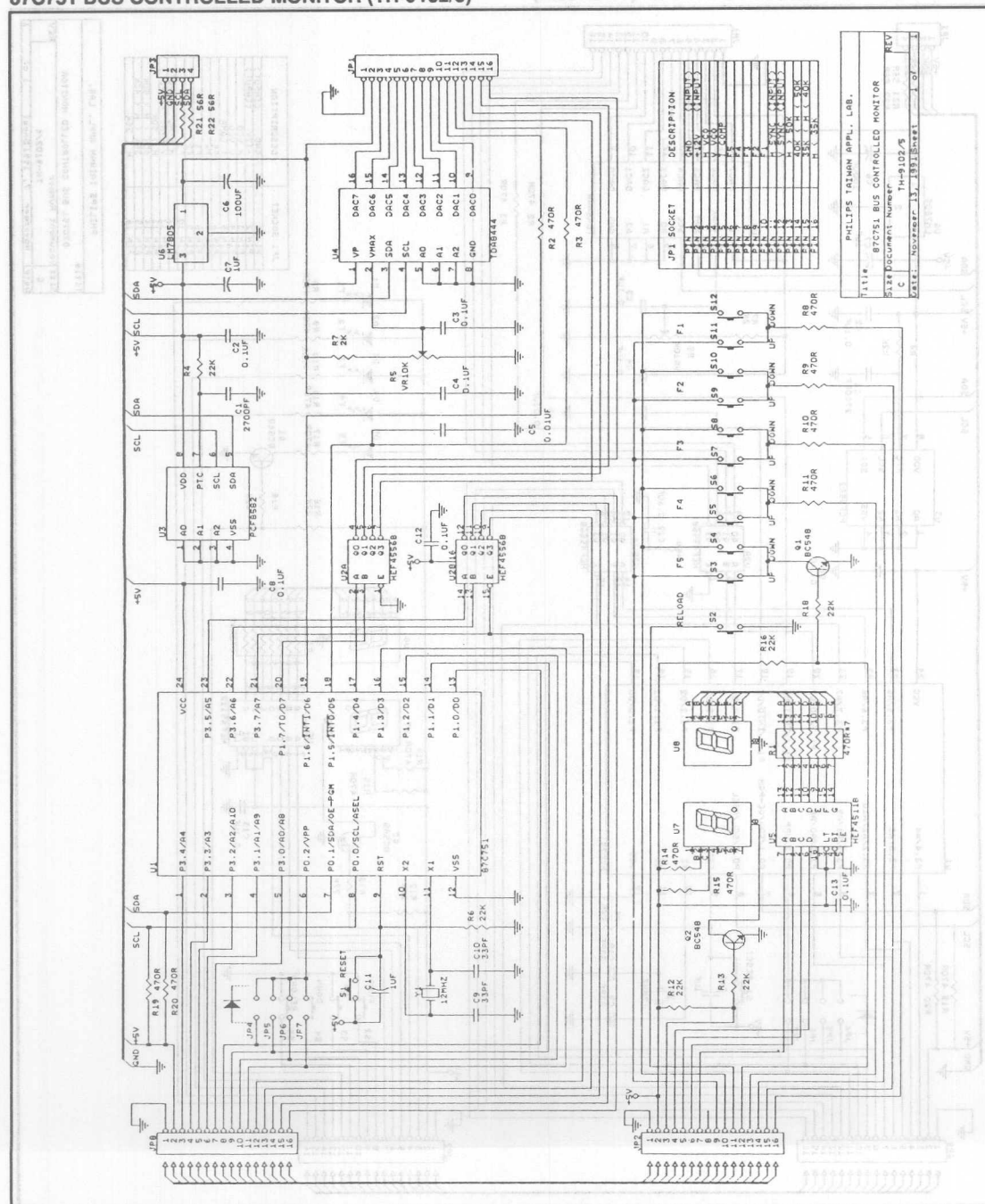
AN442

(BCM) 87C751

Specification for a bus-controlled monitor

AN442

87C751 BUS CONTROLLED MONITOR (TH-9102/5)



(BCM) 87C751
Specification for a bus-controlled monitor

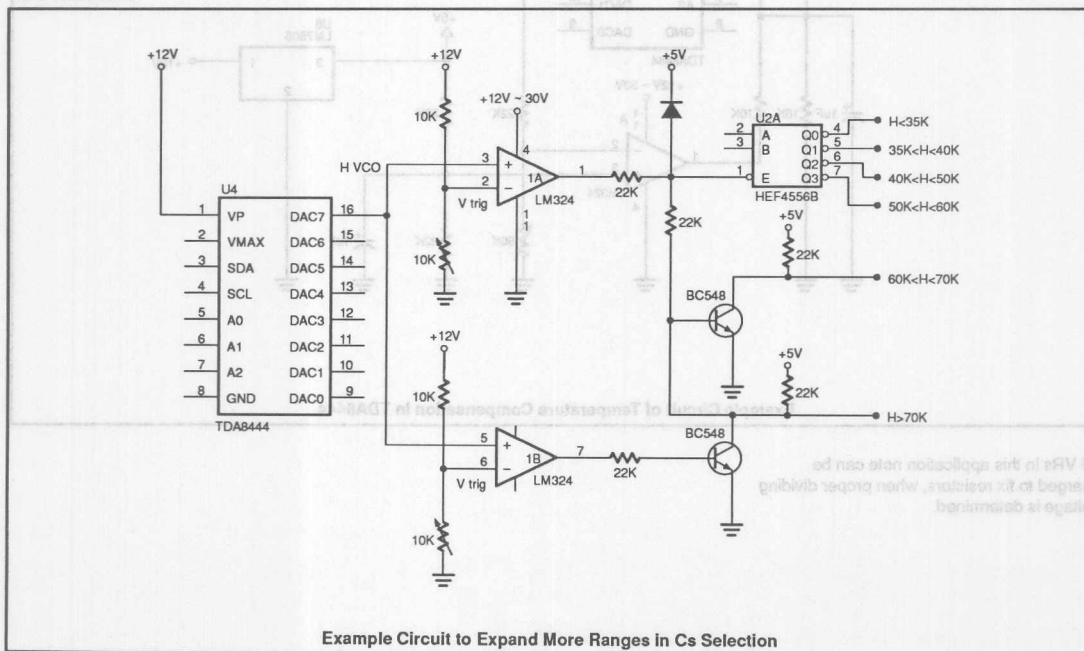
AN442

APPENDIX A

References

- | | | |
|---|---|--|
| 1. Philips Data Book IC02a, IC02b
<i>Video and Associated Systems</i> | <i>RF Communications</i>
Title: "AN168: The Inter-Integrated (I ² C) Serial Bus: Theory and Practical Consideration",
Author: Carl Fengner | TDA8433 with I ² C Control"
Author: DJA Teuling |
| 2. Philips Data Book IC20
<i>80C51 and Derivative Microcontrollers</i>
Title: "AN422: Using the 8XC751 Microcontroller as an I ² C Bus Master" | | 5. Title: "ETV89008: VGA Monitor with the High Resolution Colour Tube M34ECL10X36"
Author: H. Nerhees |

APPENDIX B



When you want to determine Vtrig signal, please disconnect Pulse Signal Generator (P.S.G.) Output to H. Sync demoboard from monitor and connect, and use input, set P.S.G. to 60kHz, TTL level output, then

power on the demoboard, the mode display should display "19", the the voltage in DAC H-V output pin is the trigger level voltage of 60kHz

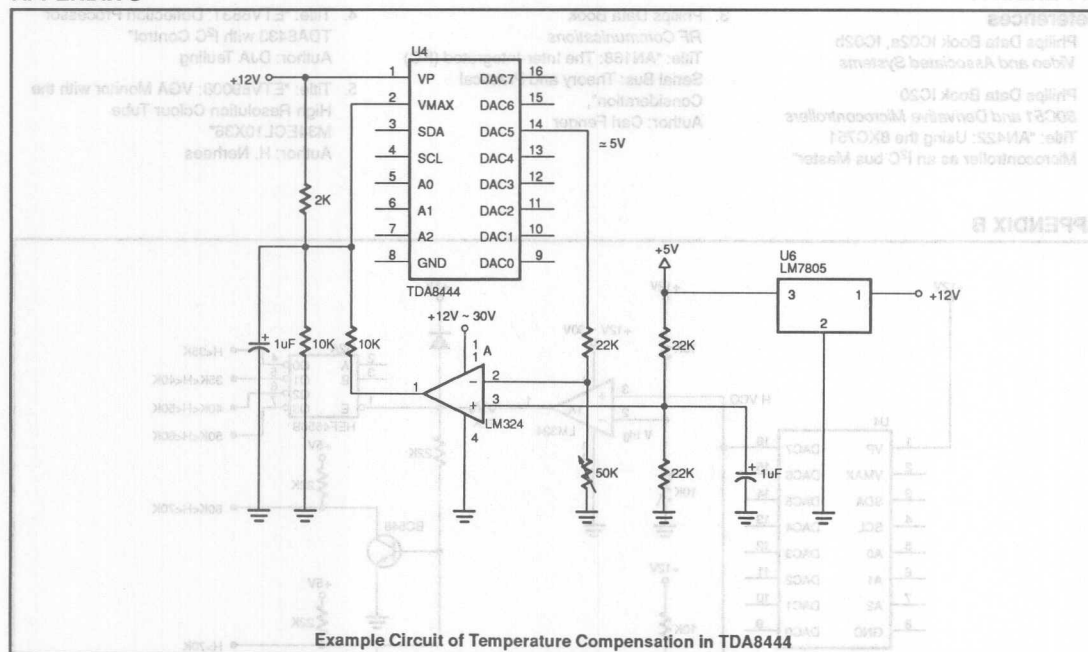
You can set P.S.G to 70kHz to measure trigger level voltage of 70kHz.

(BCM) 87C751

Specification for a bus-controlled monitor

AN442

APPENDIX C



All VRs in this application note can be charged to fix resistors, when proper dividing voltage is determined.

ACCESS.bus mouse application code for the microcontroller

AN445

DESCRIPTION

Access.bus is an open standard, defining a system for connecting a number of relatively low speed peripheral devices to a host computer, typically a desktop system. The Access.bus (A.b) standard is driven by the increasing demand of workstation and PC users for more peripherals on the desktop than ever before. Devices range from keyboards, mice and trackballs to hand held scanners, card readers and 'virtual reality' gloves. Some of the problems the A.b standard addresses are: difficulty of linking peripherals by non-expert users, desktop wiring clutter, limited number of I/O ports on a workstation, peripheral compatibility with different platforms and the high cost of software driver development associated with adding new peripherals to a system.

At the hardware level, the A.b is based on the I²C serial bus developed by Philips. The I²C protocol is supported by standard IC components, including a range of microcontrollers of the 80C51 family. These microcontrollers provide the intelligence for executing the A.b protocol in both peripheral devices and host systems. Many desktop peripherals can be implemented with a single, low cost 8XC751 microcontroller where the firmware supports both the I/O activity and the A.b protocol implementation.

This application note shows the 8XC751 firmware of Digital Equipment Corporation's A.b mouse implementation. Many A.b desktop devices could be implemented with a very similar code. After some discussion of mouse operation we shall give a short overview of the A.b protocol. Our discussion of the A.b is by no means complete—please refer to the specifications for more detailed information.

MOUSE OPERATION

The mouse is the most popular pointing device for interactive operation with a workstation, personal computer or Windows terminal. It reports to the host two dimensional planar movement, and user's activation of two or three buttons.

Many of the mice available today are opto-mechanical, using shaft encoders. As the mouse is moved over its pad, a lightweight rubber ball turns two perpendicular shafts. When the mouse is held with its cable at the top (away from the user), a left-right movement will rotate the 'X' shaft and an up-down movement will rotate the 'Y' shaft. Any diagonal movement will affect both. The shafts rotate slotted encoder disks which intercept light emitted by an LED. For

each shaft there are two phototransistors detecting the light, producing two signals which are out of phase by 90 degrees.

Figure 1 shows the waveforms produced for one of the shafts when it rotates. The changes in these quadrature signals can be detected to determine the direction of the mouse movement, and its magnitude. The "positive movement" waveforms relate, for example, to a left to right movement in the X direction. Denoting channel samples as 'AB', a transition from a '00' state to '10' shows a positive movement, while a transition from '00' to '01' shows a negative movement.

The resolution of a mouse is determined by the number of changes to the quadrature waveforms produced in a unit length of planar movement. This is determined by the mechanics of the mouse, regardless of the speed in which the mouse is being moved. The mouse is an incremental pointing device, giving the host periodical position reports which show the displacement change relative to the last report. The microcontroller in the mouse takes the burden of keeping track of the rapid quadrature waveform changes and computing the relative displacement accumulated for each new position report. The quadrature waveforms are sampled, the changes are determined to be positive or negative, and X and Y relative displacement accumulators are being incremented or decremented accordingly.

The average rate of change is determined by the speed of mouse movement. For accurate position reports the encoder waveforms should be sampled frequently enough in order not to miss changes. The DEC mouse produces 200 changes for one inch of movement. Mouse movement at 10 inches per second will yield event rate of 2000 per second, and the microcontroller attempts to sample the encoder waveforms with at least twice that rate—no more than 250 μ S between samples. The MAIN routine of the example program performs this sampling in an infinite loop. It reads the position detectors at port 3, compares it to prior readings and if there was a change computes the new value of the relative displacement accumulators YCOUNT and XCOUNT.

Position reports are sent to the host at a much slower rate. In this example, Timer0 interrupts the code at the reporting intervals, and its interrupt routine ("Timer0") initiates a message transmission to the host with the latest information if there was some change in the mouse position or the buttons. The Timer0 service routine samples the position of the three mouse buttons sensed on port 1.

Button changes are reported to the host in the same message as the position reports.

ACCESS.BUS PROTOCOL OVERVIEW

The A.b communications protocol is layered in three levels. The lowest level is a subset of the Philips Inter-integrated Circuit (I²C) bus protocol, above it the A.b Base Protocol common to all types of A.b devices, and on top are the Application Protocols which define message semantics that are specific to particular functional types of devices.

The I²C protocol defines the low level transaction over the I²C serial bus, using a single data line (SDA) and a clock line (SCL). The hardware definition for the A.b includes a four wire cable comprised of SDA, SCL and two voltage supply lines. The I²C provides for cooperative synchronization of the serial clock, bus arbitration, addressing, byte framing and byte acknowledgement by the receiver. The I²C is a multimaster protocol, and in ACCESS.bus subset the transmitter is always a master. The I²C allows 128 7 bit addresses, of which 125 may be used in A.b for peripheral microcontrollers. The I²C protocol burden is typically handled by microcontrollers both at the peripherals and at the host.

The Base Protocol establishes the A.b characteristics including message envelope format, predefined control and status messages, configuration process and the special role of the host. The host acts as a manager of the bus, and all data communication is between the host and peripheral devices—there are no message transactions between peripherals. In A.b, masters are exclusively senders and slaves are exclusively receivers. The host and the attached devices assume master or slave roles at the proper time.

An A.b message is an I²C bus transaction—a string of bytes sent by a master transmitter where each byte is acknowledged by the slave receiver, and the whole transaction is delimited by Start and Stop conditions. The minimum length of a message is four bytes, and the format definition includes specific locations for source address, destination address, message length and checksum. A protocol flag bit specifies whether the message is a device data stream or a control/status message.

The configuration process is designed to permit auto addressing and hot-plugging of devices. This process detects what devices

ACCESS.bus mouse application code for the microcontroller

AN445

are present on the bus, assigns unique bus addresses to the attached devices and connects them with the appropriate bus drivers. The configuration process is supported by eight pre-defined control/status messages. In any A.b system the host address is always the same (50H). When the system is powered up all the peripherals perform self testing, assume a default address (6EH) and send to the host an Attention message announcing their presence. The host sends to each device an identification request message, to which the devices respond with a unique 28 byte ID string. Having received the ID string, the host assigns to each device a unique address. In the case of hot-plugging, the peripheral device and the host will interact in a manner similar to the message exchange during system power up.

In the last phase of the configuration process the host interrogates each device for its "capabilities string"—which describes the functional characteristics and the potential operating modes of the A.b peripheral. Capabilities information allows the software to recognize and use bus devices without additional knowledge about their specific implementation. Using the capabilities information enables writing 'generic' software drivers that can support a range of similar devices, providing some level of device independence and modularity. The capabilities information is transferred to the host as a readable ASCII string with a simple syntax.

A.b application protocols are specific to particular types of devices. The initial A.b specifications define Application protocols for three classes of peripherals: keyboards, locators and text devices. Each class is relatively broadly defined, leaving room for a variety of different devices. When drivers in the operating software of the host fully support a certain class, all devices conforming to the relevant Application Protocol will be supported, without any need for a special software driver.

The Application protocols already defined can support many standard desktop peripherals. The Keyboards protocol supports up to 255 keys. Locator devices can have up to 15 degrees of freedom and up to 16 binary keys or buttons. This can cover devices like mouse, tablet, trackball, 'virtual reality' pointing gloves, dial boxes and function key boxes. The Text Device protocol supports devices that transmit or receive messages consisting of strings of characters in some fixed character set. The simple protocol allows high level flow control, and is

appropriate for devices like barcode readers, printers and magnetic card readers.

Each of the Application Protocols has its own set of control/status messages in addition to the predefined messages of the Base Protocol.

I²C Protocol Handling

The I²C hardware interface on the 8XC751 operates on a bit by bit basis, and the full I²C protocol is supported by a combination of hardware and firmware. This arrangement results in a very compact hardware circuitry necessary for a low cost integrated circuit. The hardware activates and monitors the SDA and SCL lines, performs the necessary arbitration and framing errors checks, and takes care of clock stretching and synchronization. The hardware is synchronized to the software either through polled loops or interrupts. An I²C interrupt is usually requested (if enabled) when a rising edge of SCL indicates a new data on the bus (DRDY), or when a special condition occurs: a frame Start (STR), Stop (STP) or an arbitration loss (ARL). The interrupt is caused by the ATN flag, which is turned on by any of the interrupt inducing conditions. The ATN flag can be polled in a software loop as well.

The example code handles the I²C protocol from an interrupt service routine (ISR). Typically, processing of a frame will be started with an interrupt (at the I2CINT label). If the bus operates at full speed, firmware processing inside a frame will be synchronized to the hardware bit by bit by a polling loop. The firmware polls the ATN flag in a loop limited to about 50 us (WaitATN) whenever it expects something to happen on the bus. If nothing happens during this period of time, the ISR is exited with the I²C interrupt re-enabled. When some bus event will occur later on, processing will resume with a new interrupt.

Processing of bus events monitored by the polling loop is identical to processing events detected by an interrupt. The context from which the mouse was sending or receiving a message is maintained between events (ATN flag activations), and is not lost when exiting the interrupt service routine. The I2Ccx byte stores the event that is expected, like waiting to send a bit or waiting for an acknowledge. Other I²C context elements are the data byte currently in the send or receive process (I2CDat), a bit counter (BitCnt) keeping track of the location within that data byte and a message byte counter (ByteCnt).

In addition to the parameters that maintain the context of the very 'generic' I²C

communications, the code maintains some additional context elements that are relevant to the higher level A.b protocol. These are the computed checksum (Check), the type of message or command being received (RcvType), the type of message being sent or pending (SndType) and a flag indicating that a Position Report transmission is pending (SendRpt).

The Interrupt service routine proceeds in handling the low level details of the I²C protocol as a Slave receiver or a Master transmitter. The routines for Slave or Master processing are separate, and the jump to either one from DISPAT in I2CDONE routine is determined by the MST bit of the I2CON hardware register. The code examines the flags determining which event caused the ATN and then handles the low level hardware according to the context, performing actions like reading a new bit, acknowledging, sending a bit, issuing a Stop and so forth.

When the low level slave receiver code completes reception of a byte, it calls the DORXB routine which deals with the contents of the byte—"application level" routine. Upon return from DORXB there is a call to the Sample subroutines. We effectively sample the quadrature waveforms in between I²C words in order to comply with the requirement for minimum sampling interval. It is interesting to note that code design does not completely separate application code from the I²C low level code—we call Sample from an I²C reception routine (and we do the same in the Master transmission routine). This is because in the 8XC751 the I²C bit processing cannot be done in parallel to other firmware activities and we have to make sure that the application's timing requirements are not being violated.

The Master code will start sending a message if the processing routine was entered due to a Start condition. The routine, in fact, fulfills a request that was issued somewhere else in the code. For example, Timer 0 ISR sets the MASTRQ bit of the I2CON register, and sets the SendRpt flag. MASTRQ causes the processor to seize the bus when it is free and issue a Start. The Master processing routine examines the SendRpt flag, and if it is set the routine will start sending a Position Report.

In a structure similar to the Slave code bit level details are handled in the MASTER routine. Byte transmissions are set up in the DOTXB routine.

ACCESS.bus mouse application code for the microcontroller

AN445

Processing At The ACCESS.bus Protocol Level

A control/status message from the host is identified by the Protocol Flag, the most significant bit of the third message byte. The message body is a code for the command. When such messages are received, they are processed by the DORXCMD routine. Control/Status messages can be of either the Base Protocol or the Application Protocol. In the listing, base protocol codes have the prefix 'L_', and application level protocol commands has the prefix 'App_' (the definitions are in the include file 'ab.inc').

The Base Protocol commands from the host are L_Reset, L_IdReq, L_AsgnAdr and L_CapReq. During the configuration process the mouse responds to the host with device to host control/status messages: L_Attn, L_IdReply, L_CapReply and L_Error.

The string for the L_IdReply message is defined in GET_ID. The module revision and vendor name are padded with space characters in order to fit the fixed string length. The last four bytes of the ID string are a device number that can distinguish otherwise like devices with the same firmware. The protocol allows it to be a serial number or a pseudo random number. Our mouse uses a pseudo random number, produced by reading the 16 bit contents of Timer0 that is active since power-up (the number is extended to 32 bits by appending FFFF). The protocol includes 'protection'

against the rare event in which two like devices report the same pseudo random number or are mistakenly assigned to the same address. Just prior to sending the first data message to the host, each interactive device transmits a Reset message to its own assigned address (see PosMsg label in the example code). Any other device with the same address will be forced to the power-up default address and will undergo configuration again, as it was hot-plugged onto the bus.

The Capabilities String for the L_CapReply message is defined in GET_CAP. The string identifies the device as a mouse with specific characteristics: three buttons, two dimensions, relative location reports with 200 dpi resolution etc. The 'prot(locator)' element tells the A.b software driver to use the Locator Device Protocol.

The Locator Device Protocol is one of three application protocols already defined for the highest layer of the A.b protocol. This protocol defines a "Locator Event Report" which is used for the Position Reports of the mouse.

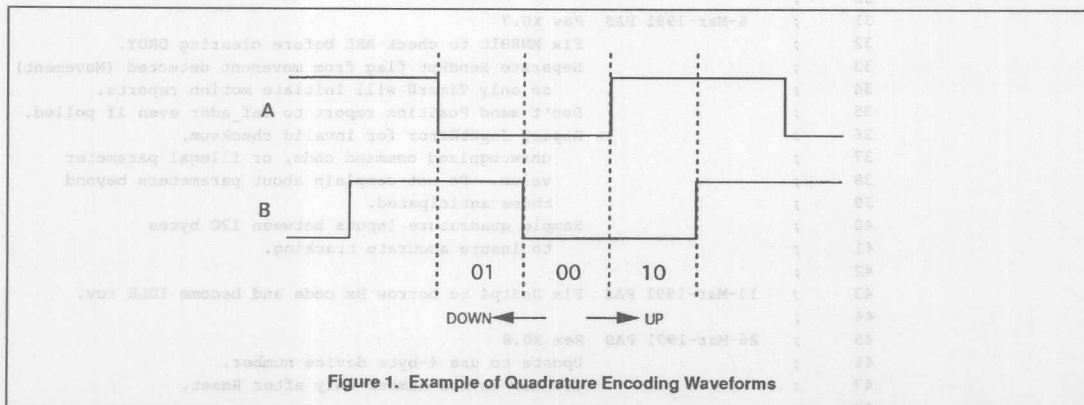
A Locator Event Report is sent in the format of the device data stream Message defined in the base A.b protocol. The message body includes the current state of the buttons and the location difference from the last report. This data is coded as a sequence of two byte integers. For the mouse which is a two dimensional device, the message data

stream length is six bytes, or three integers. The first integer contains the state of 0-15 locator key switches. For the three button mouse, only three of these sixteen bits carry meaningful information. The remaining integers represent the position of the locator dimensions—the contents of the X and Y displacement accumulators.

Three control messages specific to the Locator Protocol are processed at DORXCMD. The host initiates a self test by App_Test. App_Poll initiates one time transmission of a position report, and App_Setinterval modifies the default reporting interval controlled by the reload value of Timer0.

This note highlights some of the implementation details—the commented listing covers the rest. As one can see, the A.b protocols are relatively simple to program in firmware. The low-level I²C implementation on the 8XC751 is somewhat involved, but the same low level routines can be re-used for different devices.

The source code files for this program are available for download from the Philips Semiconductors computer bulletin board system. This system is open to all callers, operates 24 hours a day, and can be accessed with modems at 2400, 1200, and 300 baud. The telephone numbers for the BBS are: (800) 451-6644 (in the U.S. only) or (408) 991-2406.



ACCESS.bus mouse application code for the microcontroller

AN445

MS-DOS MACRO ASSEMBLER A51 V4.4
 OBJECT MODULE PLACED IN MOUSE.OBJ
 ASSEMBLER INVOKED BY: A51 MOUSE.A51
 LOC OBJ LINE SOURCE

```

1 ;*****
2 ; Module: mouse.a51
3 ;
4 ; Firmware design and code for I2C desktop bus Mouse
5 ; Environment: 83C751 Assembler
6 ; Author: Robert Clemens 10-Jul-1990
7 ; (I2C I/O adapted from P.Sichel's "Keyboard" code)
8 ; Revision: 6-Mar-1991
9 ;
10 ; 31-Jan-1991 PAS Add numerous keyboard fixes.
11 ; Streamline input sample and I2C code.
12 ; Separate HW dependent constants.
13 ; Fix RxEnable after bus time out.
14 ;
15 ; 06-Feb-1991 PAS Rev X0.6
16 ; Fix sample timer initialization, use 14ms as default.
17 ; Fix length checking to allow commands with
18 ; more parameters than required.
19 ; Implement Set Interval command.
20 ; Handle LLLLL=0 to mean 32.
21 ; Fix ARL during self-addressed reset message.
22 ; Fix to handle DRDY and ARL together.
23 ; Re-order mouse buttons as MRL, update Capabilities.
24 ; Do not allow other interrupts during TimerI svc.
25 ; Document sampling requirements.
26 ; Document hardware details.
27 ; Add check to skip waiting after DNRXB.
28 ; Misc clean-up in: BeMast, Assign,...
29 ; Use include files for 751 registers and ODB msgs.
30 ;
31 ; 6-Mar-1991 PAS Rev X0.7
32 ; Fix MN8Bit to check ARL before clearing DRDY.
33 ; Separate SendRpt flag from movement detected (Movement)
34 ; so only Timer0 will initiate motion reports.
35 ; Don't send Position report to def_addr even if polled.
36 ; Report InputError for invalid checksum,
37 ; unrecognized command code, or illegal parameter
38 ; value. Do not complain about parameters beyond
39 ; those anticipated.
40 ; Sample quadrature inputs between I2C bytes
41 ; to insure accurate tracking.
42 ;
43 ; 13-Mar-1991 PAS Fix DoStp4 to borrow Rx code and become IDLE rcv.
44 ;
45 ; 26-Mar-1991 PAS Rev X0.8
46 ; Update to use 4-byte device number.
47 ; Get new device number only after Reset.
48 ;
49 ; 8-Apr-1991 PAS Add error checking to CapReq.
50 ;
51 ; 9-Jul-1991 PAS Add protocol_revision as part of module revision.
52 ;
53 ; 29-Jul-1991 PAS Protocol_revision in 1-byte, fix ARL and MASTER bug.
54 ; Ignore unrecognized commands. Add I_MsgCheck.
55 ;
56 ; 11-Sep-1991 PAS Rev X1.2. Identify button positions as 1,2,3.

```

ACCESS.bus mouse application code for the
microcontroller

AN445

```

LOC  OBJ          LINE      SOURCE
57      ;          Use new ab.inc file.  Change I_MsgCheck to
58      ;          I_Error and do not overwrite pending SndType.
59      ;          Retry message after Negative Ack (NACnt).
60      ;          17-Oct-1991 PAS Rev X1.3.3 Fix spurious large count problem.
61      ;          Fix STOP detected while MASTER without ARL.
62      ;          25-Oct-1991 PAS Rev X1.4.0 Improve TimerI handler to avoid
63      ;          lockup when MASTRQ with SCL low.
64      ;          4-Nov-1991 PAS V1.0 release for Boston mfg.
65      ;          22-Dec-1991 PAS V1.1 align data with tx bit that lost arbitration
66      ;
67      ;
68      ;
69      ;
70      ;
71      ;
72      ;
73      ;
74      ;
75      ;
76      $ TITLE (Digital ACCESS.bus Mouse, V1.1)
77      $ DATE (12/22/91)
78      $ DEBUG
79      $ NOMOD51          ;83C751 is not model 51
80      ;Define SFRs explicitly
81      ;
82      ;
83      ; Symbolic addresses and masks
84      ;
85      $ INCLUDE( /dskbus/include/arch/reg751.inc )
86      ;*****
87      ; Module: /dskbus/include/arch/reg751.inc
88      ;
89      ; 83C751 SFR declarations
90      ; Environment: 83C751 Assembler
91      ;
92      ; Date          Revision      Perpetrator
93      ; 30-Jan-91    X0.1          Mark Shepard
94      ; Created (from previous keyboard module)
95      ;
96      ;
97      ;
98      ;
99      ;
100     ;
101     ; *****
102     ; *****
103     ; *****
104     ; *****
105     ; *****
106     ; *****
107     ; *****
108     ; *****
109     ; *****
110     ; *****
111     ; *****
112     ; *****
113     ; *****
114     ; *****
115     ; *****
116     ; *****
117     ; *****
118     ; *****
119     ; *****
120     ; *****
121     ; *****
122     ; *****
123     ; *****
124     ; *****
125     ; *****
126     ; *****
127     ; *****
128     ; *****
129     ; *****
130     ; *****
131     ; *****
132     ; *****
133     ; *****
134     ; *****
135     ; *****
136     ; *****
137     ; *****
138     ; *****
139     ; *****
140     ; *****
141     ; *****
142     ; *****
143     ; *****
144     ; *****
145     ; *****
146     ; *****
147     ; *****
148     ; *****
149     ; *****
150     ; *****
151     ; *****
152     ; *****
153     ; *****
154     ; *****
155     ; *****
156     ; *****
157     ; *****
158     ; *****
159     ; *****
160     ; *****
161     ; *****
162     ; *****
163     ; *****
164     ; *****
165     ; *****
166     ; *****
167     ; *****
168     ; *****
169     ; *****
170     ; *****
171     ; *****
172     ; *****
173     ; *****
174     ; *****
175     ; *****
176     ; *****
177     ; *****
178     ; *****
179     ; *****
180     ; *****
181     ; *****
182     ; *****
183     ; *****
184     ; *****
185     ; *****
186     ; *****
187     ; *****
188     ; *****
189     ; *****
190     ; *****
191     ; *****
192     ; *****
193     ; *****
194     ; *****
195     ; *****
196     ; *****
197     ; *****
198     ; *****
199     ; *****
200     ; *****
201     ; *****
202     ; *****
203     ; *****
204     ; *****
205     ; *****
206     ; *****
207     ; *****
208     ; *****
209     ; *****
210     ; *****
211     ; *****
212     ; *****
213     ; *****
214     ; *****
215     ; *****
216     ; *****
217     ; *****
218     ; *****
219     ; *****
220     ; *****
221     ; *****
222     ; *****
223     ; *****
224     ; *****
225     ; *****
226     ; *****
227     ; *****
228     ; *****
229     ; *****
230     ; *****
231     ; *****
232     ; *****
233     ; *****
234     ; *****
235     ; *****
236     ; *****
237     ; *****
238     ; *****
239     ; *****
240     ; *****
241     ; *****
242     ; *****
243     ; *****
244     ; *****
245     ; *****
246     ; *****
247     ; *****
248     ; *****
249     ; *****
250     ; *****
251     ; *****
252     ; *****
253     ; *****
254     ; *****
255     ; *****
256     ; *****
257     ; *****
258     ; *****
259     ; *****
260     ; *****
261     ; *****
262     ; *****
263     ; *****
264     ; *****
265     ; *****
266     ; *****
267     ; *****
268     ; *****
269     ; *****
270     ; *****
271     ; *****
272     ; *****
273     ; *****
274     ; *****
275     ; *****
276     ; *****
277     ; *****
278     ; *****
279     ; *****
280     ; *****
281     ; *****
282     ; *****
283     ; *****
284     ; *****
285     ; *****
286     ; *****
287     ; *****
288     ; *****
289     ; *****
290     ; *****
291     ; *****
292     ; *****
293     ; *****
294     ; *****
295     ; *****
296     ; *****
297     ; *****
298     ; *****
299     ; *****
300     ; *****
301     ; *****
302     ; *****
303     ; *****
304     ; *****
305     ; *****
306     ; *****
307     ; *****
308     ; *****
309     ; *****
310     ; *****
311     ; *****
312     ; *****
313     ; *****
314     ; *****
315     ; *****
316     ; *****
317     ; *****
318     ; *****
319     ; *****
320     ; *****
321     ; *****
322     ; *****
323     ; *****
324     ; *****
325     ; *****
326     ; *****
327     ; *****
328     ; *****
329     ; *****
330     ; *****
331     ; *****
332     ; *****
333     ; *****
334     ; *****
335     ; *****
336     ; *****
337     ; *****
338     ; *****
339     ; *****
340     ; *****
341     ; *****
342     ; *****
343     ; *****
344     ; *****
345     ; *****
346     ; *****
347     ; *****
348     ; *****
349     ; *****
350     ; *****
351     ; *****
352     ; *****
353     ; *****
354     ; *****
355     ; *****
356     ; *****
357     ; *****
358     ; *****
359     ; *****
360     ; *****
361     ; *****
362     ; *****
363     ; *****
364     ; *****
365     ; *****
366     ; *****
367     ; *****
368     ; *****
369     ; *****
370     ; *****
371     ; *****
372     ; *****
373     ; *****
374     ; *****
375     ; *****
376     ; *****
377     ; *****
378     ; *****
379     ; *****
380     ; *****
381     ; *****
382     ; *****
383     ; *****
384     ; *****
385     ; *****
386     ; *****
387     ; *****
388     ; *****
389     ; *****
390     ; *****
391     ; *****
392     ; *****
393     ; *****
394     ; *****
395     ; *****
396     ; *****
397     ; *****
398     ; *****
399     ; *****
400     ; *****
401     ; *****
402     ; *****
403     ; *****
404     ; *****
405     ; *****
406     ; *****
407     ; *****
408     ; *****
409     ; *****
410     ; *****
411     ; *****
412     ; *****
413     ; *****
414     ; *****
415     ; *****
416     ; *****
417     ; *****
418     ; *****
419     ; *****
420     ; *****
421     ; *****
422     ; *****
423     ; *****
424     ; *****
425     ; *****
426     ; *****
427     ; *****
428     ; *****
429     ; *****
430     ; *****
431     ; *****
432     ; *****
433     ; *****
434     ; *****
435     ; *****
436     ; *****
437     ; *****
438     ; *****
439     ; *****
440     ; *****
441     ; *****
442     ; *****
443     ; *****
444     ; *****
445     ; *****
446     ; *****
447     ; *****
448     ; *****
449     ; *****
450     ; *****
451     ; *****
452     ; *****
453     ; *****
454     ; *****
455     ; *****
456     ; *****
457     ; *****
458     ; *****
459     ; *****
460     ; *****
461     ; *****
462     ; *****
463     ; *****
464     ; *****
465     ; *****
466     ; *****
467     ; *****
468     ; *****
469     ; *****
470     ; *****
471     ; *****
472     ; *****
473     ; *****
474     ; *****
475     ; *****
476     ; *****
477     ; *****
478     ; *****
479     ; *****
480     ; *****
481     ; *****
482     ; *****
483     ; *****
484     ; *****
485     ; *****
486     ; *****
487     ; *****
488     ; *****
489     ; *****
490     ; *****
491     ; *****
492     ; *****
493     ; *****
494     ; *****
495     ; *****
496     ; *****
497     ; *****
498     ; *****
499     ; *****
500     ; *****
501     ; *****
502     ; *****
503     ; *****
504     ; *****
505     ; *****
506     ; *****
507     ; *****
508     ; *****
509     ; *****
510     ; *****
511     ; *****
512     ; *****
513     ; *****
514     ; *****
515     ; *****
516     ; *****
517     ; *****
518     ; *****
519     ; *****
520     ; *****
521     ; *****
522     ; *****
523     ; *****
524     ; *****
525     ; *****
526     ; *****
527     ; *****
528     ; *****
529     ; *****
530     ; *****
531     ; *****
532     ; *****
533     ; *****
534     ; *****
535     ; *****
536     ; *****
537     ; *****
538     ; *****
539     ; *****
540     ; *****
541     ; *****
542     ; *****
543     ; *****
544     ; *****
545     ; *****
546     ; *****
547     ; *****
548     ; *****
549     ; *****
550     ; *****
551     ; *****
552     ; *****
553     ; *****
554     ; *****
555     ; *****
556     ; *****
557     ; *****
558     ; *****
559     ; *****
560     ; *****
561     ; *****
562     ; *****
563     ; *****
564     ; *****
565     ; *****
566     ; *****
567     ; *****
568     ; *****
569     ; *****
570     ; *****
571     ; *****
572     ; *****
573     ; *****
574     ; *****
575     ; *****
576     ; *****
577     ; *****
578     ; *****
579     ; *****
580     ; *****
581     ; *****
582     ; *****
583     ; *****
584     ; *****
585     ; *****
586     ; *****
587     ; *****
588     ; *****
589     ; *****
590     ; *****
591     ; *****
592     ; *****
593     ; *****
594     ; *****
595     ; *****
596     ; *****
597     ; *****
598     ; *****
599     ; *****
600     ; *****
601     ; *****
602     ; *****
603     ; *****
604     ; *****
605     ; *****
606     ; *****
607     ; *****
608     ; *****
609     ; *****
610     ; *****
611     ; *****
612     ; *****
613     ; *****
614     ; *****
615     ; *****
616     ; *****
617     ; *****
618     ; *****
619     ; *****
620     ; *****
621     ; *****
622     ; *****
623     ; *****
624     ; *****
625     ; *****
626     ; *****
627     ; *****
628     ; *****
629     ; *****
630     ; *****
631     ; *****
632     ; *****
633     ; *****
634     ; *****
635     ; *****
636     ; *****
637     ; *****
638     ; *****
639     ; *****
640     ; *****
641     ; *****
642     ; *****
643     ; *****
644     ; *****
645     ; *****
646     ; *****
647     ; *****
648     ; *****
649     ; *****
650     ; *****
651     ; *****
652     ; *****
653     ; *****
654     ; *****
655     ; *****
656     ; *****
657     ; *****
658     ; *****
659     ; *****
660     ; *****
661     ; *****
662     ; *****
663     ; *****
664     ; *****
665     ; *****
666     ; *****
667     ; *****
668     ; *****
669     ; *****
670     ; *****
671     ; *****
672     ; *****
673     ; *****
674     ; *****
675     ; *****
676     ; *****
677     ; *****
678     ; *****
679     ; *****
680     ; *****
681     ; *****
682     ; *****
683     ; *****
684     ; *****
685     ; *****
686     ; *****
687     ; *****
688     ; *****
689     ; *****
690     ; *****
691     ; *****
692     ; *****
693     ; *****
694     ; *****
695     ; *****
696     ; *****
697     ; *****
698     ; *****
699     ; *****
700     ; *****
701     ; *****
702     ; *****
703     ; *****
704     ; *****
705     ; *****
706     ; *****
707     ; *****
708     ; *****
709     ; *****
710     ; *****
711     ; *****
712     ; *****
713     ; *****
714     ; *****
715     ; *****
716     ; *****
717     ; *****
718     ; *****
719     ; *****
720     ; *****
721     ; *****
722     ; *****
723     ; *****
724     ; *****
725     ; *****
726     ; *****
727     ; *****
728     ; *****
729     ; *****
730     ; *****
731     ; *****
732     ; *****
733     ; *****
734     ; *****
735     ; *****
736     ; *****
737     ; *****
738     ; *****
739     ; *****
740     ; *****
741     ; *****
742     ; *****
743     ; *****
744     ; *****
745     ; *****
746     ; *****
747     ; *****
748     ; *****
749     ; *****
750     ; *****
751     ; *****
752     ; *****
753     ; *****
754     ; *****
755     ; *****
756     ; *****
757     ; *****
758     ; *****
759     ; *****
760     ; *****
761     ; *****
762     ; *****
763     ; *****
764     ; *****
765     ; *****
766     ; *****
767     ; *****
768     ; *****
769     ; *****
770     ; *****
771     ; *****
772     ; *****
773     ; *****
774     ; *****
775     ; *****
776     ; *****
777     ; *****
778     ; *****
779     ; *****
780     ; *****
781     ; *****
782     ; *****
783     ; *****
784     ; *****
785     ; *****
786     ; *****
787     ; *****
788     ; *****
789     ; *****
790     ; *****
791     ; *****
792     ; *****
793     ; *****
794     ; *****
795     ; *****
796     ; *****
797     ; *****
798     ; *****
799     ; *****
800     ; *****
801     ; *****
802     ; *****
803     ; *****
804     ; *****
805     ; *****
806     ; *****
807     ; *****
808     ; *****
809     ; *****
810     ; *****
811     ; *****
812     ; *****
813     ; *****
814     ; *****
815     ; *****
816     ; *****
817     ; *****
818     ; *****
819     ; *****
820     ; *****
821     ; *****
822     ; *****
823     ; *****
824     ; *****
825     ; *****
826     ; *****
827     ; *****
828     ; *****
829     ; *****
830     ; *****
831     ; *****
832     ; *****
833     ; *****
834     ; *****
835     ; *****
836     ; *****
837     ; *****
838     ; *****
839     ; *****
840     ; *****
841     ; *****
842     ; *****
843     ; *****
844     ; *****
845     ; *****
846     ; *****
847     ; *****
848     ; *****
849     ; *****
850     ; *****
851     ; *****
852     ; *****
853     ; *****
854     ; *****
855     ; *****
856     ; *****
857     ; *****
858     ; *****
859     ; *****
860     ; *****
861     ; *****
862     ; *****
863     ; *****
864     ; *****
865     ; *****
866     ; *****
867     ; *****
868     ; *****
869     ; *****
870     ; *****
871     ; *****
872     ; *****
873     ; *****
874     ; *****
875     ; *****
876     ; *****
877     ; *****
878     ; *****
879     ; *****
880     ; *****
881     ; *****
882     ; *****
883     ; *****
884     ; *****
885     ; *****
886     ; *****
887     ; *****
888     ; *****
889     ; *****
890     ; *****
891     ; *****
892     ; *****
893     ; *****
894     ; *****
895     ; *****
896     ; *****
897     ; *****
898     ; *****
899     ; *****
900     ; *****
901     ; *****
902     ; *****
903     ; *****
904     ; *****
905     ; *****
906     ; *****
907     ; *****
908     ; *****
909     ; *****
910     ; *****
911     ; *****
912     ; *****
913     ; *****
914     ; *****
915     ; *****
916     ; *****
917     ; *****
918     ; *****
919     ; *****
920     ; *****
921     ; *****
922     ; *****
923     ; *****
924     ; *****
925     ; *****
926     ; *****
927     ; *****
928     ; *****
929     ; *****
930     ; *****
931     ; *****
932     ; *****
933     ; *****
934     ; *****
935     ; *****
936     ; *****
937     ; *****
938     ; *****
939     ; *****
940     ; *****
941     ; *****
942     ; *****
943     ; *****
944     ; *****
945     ; *****
946     ; *****
947     ; *****
948     ; *****
949     ; *****
950     ; *****
951     ; *****
952     ; *****
953     ; *****
954     ; *****
955     ; *****
956     ; *****
957     ; *****
958     ; *****
959     ; *****
960     ; *****
961     ; *****
962     ; *****
963     ; *****
964     ; *****
965     ; *****
966     ; *****
967     ; *****
968     ; *****
969     ; *****
970     ; *****
971     ; *****
972     ; *****
973     ; *****
974     ; *****
975     ; *****
976     ; *****
977     ; *****
978     ; *****
979     ; *****
980     ; *****
981     ; *****
982     ; *****
983     ; *****
984     ; *****
985     ; *****
986     ; *****
987     ; *****
988     ; *****
989     ; *****
990     ; *****
991     ; *****
992     ; *****
993     ; *****
994     ; *****
995     ; *****
996     ; *****
997     ; *****
998     ; *****
999     ; *****
1000    ; *****

```

ACCESS.bus mouse application code for the microcontroller

AN445

```

LOC OBJ      LINE      SOURCE
0001          =1 123 1 MST EQU 1 ;master
          =1 124 2 ; Output (write) is binary values for MOV I2CON, #...
0080          =1 125 3 CXAP EQU 80h ;clear xmit active
0040          =1 126 4 IDLE EQU 40h ;set to idle slave
0020          =1 127 5 CDR EQU 20h ;clear data ready
0010          =1 128 6 CARL EQU 10h ;clear arbitration loss
0008          =1 129 7 CSTR EQU 08h ;clear start
0004          =1 130 8 CSTP EQU 04h ;clear stop
0002          =1 131 9 XSTR EQU 02h ;transmit start
0001          =1 132 10 XSTP EQU 01h ;transmit stop
          =1 133 11
          =1 134 12 ; I2C Data Register
0099          =1 135 13 I2DAT EQU 099h
0080          =1 136 14 XDAT EQU 80h ;transmit data
          =1 137 15
          =1 138 16 ; I2C Configuration Register
00D8          =1 139 17 I2CFG EQU 0D8h
0080          =1 140 18 SLAVEN EQU 80h ;enable slave mode
0007          =1 141 19 SLAVENB EQU 7 ;
0040          =1 142 20 MASTRQ EQU 40h ;master request
0006          =1 143 21 MASTRQB EQU 6 ;
0020          =1 144 22 CLRTI EQU 20h ;clear timerI
0005          =1 145 23 CLRTIB EQU 5 ;
0010          =1 146 24 TIRUN EQU 10h ;timerI run
0004          =1 147 25 TIRUNB EQU 4 ;
          =1 148 26
0088          =1 149 27 TCON EQU 088h ;Timer Control
          =1 150 28
          =1 151 29 ; 83C751 SFRs
0080          =1 152 30 P0 EQU 080h
0080          =1 153 31 SCL BIT P0.0
0081          =1 154 32 SDA BIT P0.1
0081          =1 155 33 SP EQU 081h
0082          =1 156 34 DPL EQU 082h
0083          =1 157 35 DPH EQU 083h
008A          =1 158 36 TL EQU 08Ah ;Timer Lo
008B          =1 159 37 RTL EQU 08Bh ;Reload TL
008C          =1 160 38 TH EQU 08Ch ;Timer Hi
008D          =1 161 39 RTH EQU 08Dh ;Reload TH
0090          =1 162 40 P1 EQU 090h
00B0          =1 163 41 P3 EQU 0B0h
00D0          =1 164 42 PSW EQU 0D0h
00E0          =1 165 43 ACC EQU 0E0h
00F0          =1 166 44 B EQU 0F0h
          =1 167 45
          =1 168 46 ;***** End of include file
          =1 169 47
          =1 170 48 $ INCLUDE( /diskbus/include/ab.inc )
          =1 171 49 ;*****module ab.inc*****
          =1 172 50 ;
          =1 173 51 ; Ab Base Protocol definitions
          =1 174 52 ; Environment: 83C751 Assembler
          =1 175 53 ;
          =1 179 54 ; Date Revision Perpetrator
          =1 180 55 ;
          =1 181 56 ; 04-Sep-91 ---- Mark Shepard
          =1 182 57 ; Changed I_MsgCheck to I_Error, kept I_MsgCheck for backpatibility.
          =1 183 58 ; Added App_Error (0xb4 to correspond to I_Error).
          =1 184 59 ; Changed Sig_Attn from 0 to 3 to make KB code easier.

```

ACCESS.bus mouse application code for the microcontroller

AN445

```

LOC  OBJ          LINE    SOURCE
=1 185            ;
=1 186            ; 04-Aug-91 ---- Mark Shepard
=1 187            ; Renamed to generic "ab.inc"
=1 188            ; Added defines for header structure (mainly for documentation purposes).
=1 189            ; Added defines for bus signal codes (RESET, HALT, ATTN, etc.)
=1 190            ;
=1 191            ; 22-May-91 X0.2 Peter Sichel
=1 192            ; Added Vendor command codes, and I_MsgCheck.
=1 193            ;
=1 194            ; 30-Jan-91 X0.1 Mark Shepard
=1 195            ; Created from ODB base protocol spec, X0.7.
=1 196            ;
=1 197            ;
=1 198            $EJ
=1 199            ;*****
=1 200            ; Desktop bus command codes for all Interface-part and Application-part
=1 201            ; commands in the Base Protocol spec.
=1 202            ;
=1 203            ; Naming Convention -
=1 204            ; Interface-Part codes are prefixed with "I_",
=1 205            ; Application-Part with "App_". Codes specific to a particular
=1 206            ; sub-protocol (e.g. Keyboard Protocol) could be prefixed with
=1 207            ; "Key_", "Kb_", or "Kbp_".
=1 208            ;
=1 209            ; Definitions for sub-protocols should go in separate include files.
=1 210            ; (keyp.inc, locp.inc, textp.inc, etc.).
=1 211            ;
0000 =1 212            I_NoMsg EQU 000h ; special value, means not a valid message
=1 213            ;*****
00F0 =1 214            I_Reset EQU 0f0h ; reset device
00F1 =1 215            I_IdReq EQU 0f1h ; identify request
00F2 =1 216            I_AsgnAdr EQU 0f2h ; assign address
00F3 =1 217            I_CapReq EQU 0f3h ; capabilities (fragment) request
=1 218            ;
00E0 =1 219            I_Attn EQU 0e0h ; power-on/reset attention
00E1 =1 220            I_IdReply EQU 0e1h ; identify reply
00E3 =1 221            I_CapReply EQU 0e3h ; capabilities (fragment) reply
00E4 =1 222            I_Error EQU 0e4h ; (the future) interface/bus std error report
00E4 =1 223            I_MsgCheck EQU 0e4h ; *** obsolete, don't use in new code ***
=1 224            ;
00C0 =1 225            I_Vendor0 EQU 0c0h ; vendor reserved command
00C1 =1 226            I_Vendor1 EQU 0c1h ; vendor reserved command
00C2 =1 227            I_Vendor2 EQU 0c2h ; vendor reserved command
00C3 =1 228            I_Vendor3 EQU 0c3h ; vendor reserved command
=1 229            ;
00A0 =1 230            App_Sig EQU 0a0h ; hardware signal
00A1 =1 231            App_TestReply EQU 0a1h ; test reply (for a specific application-part)
=1 232            ;
00B1 =1 233            App_Test EQU 0b1h ; self-test result request
00B4 =1 234            App_Error EQU 0b4h ; (the future) std application error report
=1 235            ;
=1 236            ;*****
=1 237            ; Well-known I2C addresses used by desktop bus peers
=1 238            ;
=1 239            ; I2C Address Allocation
=1 240            ; row column
=1 241            ; A6,A5,A4 / A3,A2,A1,A0 R/W=0 (write)
=1 242            ; Host: 2/8 50h
=1 243            ; Devices: 2/9-2/15, 3/0-3/7 52-6Eh (even numbers only)

```

ACCESS.bus mouse application code for the
microcontroller

AN445

```

LOC  OBJ          LINE    SOURCE
                                6Eh
=1 244 ; Device default: 3/7
=1 245
0050 =1 246   Adr_Host EQU 0050h ; standard Host address
006E =1 247   Adr_Default EQU 006eh ; default (pwr-up) address for peripherals
=1 248
=1 249 ;*****
=1 250 ; Bus-Signal codes defined at the Base Protocol level
=1 251
0001 =1 252   Sig_Reset EQU 1 ; hardware reset!
0002 =1 253   Sig_Halt EQU 2 ; debugging interrupt
0003 =1 254   Sig_Attn EQU 3 ; general-purpose interrupt from User
=1 255
=1 256 ;*****
=1 257 ; Defines related to message structure
=1 258 ;
=1 259 ; Declarations prefixed with "AbWire_" refer to Ab objects, fields, etc.
=1 260 ; as transmitted across the i2c "wire" (as opposed to the "optimized frame
=1 261 ; format used between the host and 83c751 host-ctrlr).
=1 262 ;
=1 263 ;
0003 =1 264   AbWire_HdrSiz EQU 3 ; Ab Header Size on the wire: dst + src + Plen
007F =1 265   AbWire_LenMask EQU 7fh ; Ab mask for length field
007F =1 266   AbWire_MaxLen EQU 127 ; Ab maximum data bytes in message
=1 267
=1 268 ;***** End of include file
=1 269
=1 270
=1 271 $EJ
=1 272 ;*****
=1 273 ;* Hardware/timing-dependent constants *
=1 274 ;*****
=1 275
=1 276   CT1 EQU 02h ;CT1, CT0 fmax = 16.8 MHz
=1 277   CT EQU 01h ;CT1, CT0 fmax = 14.25 MHz
=1 278   CT EQU 00h ;CT1, CT0 fmax = 11.7 MHz
0003 =1 279   CT EQU 03h ;CT1, CT0 fmax = 9.14 MHz
=1 280
009A =1 281   IntEnab EQU 09Ah ;enable EA+EI2+ET1+ET0.
0010 =1 282   INIT_TCON EQU 010h ;Timer0 init for internal operation.
=1 283
00DB =1 284   DEF_RTH EQU 00DBh ;Sampling interval (14ms with 8 MHz clock).
008B =1 285   DEF_RTL EQU 008Bh ; Used as default Timer0 reload value.
0002 =1 286   MSECH EQU 002h ;Timer offset for 1 ms with 8 MHz clock.
009A =1 287   MSECL EQU 009Ah ; 029Ah=666 * 3/2 mms/tick = 1000 ticks = 1ms.
=1 288
0008 =1 289   DelayATN EQU 8 ;about 50µs (wait for I2C activity)
=1 290
0010 =1 291   CapFragLen EQU 16 ;Capabilities fragment length.
=1 292
=1 293 ; *** Hardware Interface Notes ***
=1 294 ; P3 is used to read the X-Y quadrature inputs.
=1 295 ; P3.0 = XB
=1 296 ; P3.1 = XA
=1 297 ; P3.2 = YA
=1 298 ; P3.3 = YB
=1 299 ;
=1 300 ; Notice P3.0 is connected to the B side of the X encoder
=1 301 ; while P3.2 is connected to the A side of the Y encoder.
=1 302 ; This is to compensate for the orientation of the inclined cylinders.

```


ACCESS.bus mouse application code for the microcontroller

AN445

```

LOC  OBJ          LINE    SOURCE
303      ; Positive X movement causes clockwise rotation of the encoder shaft.
304      ; Positive Y movement causes counter clockwise rotation of the encoder shaft.
305      ;
306      ; P1 is used to read the switch inputs.
307      ; P1.0 = middle mouse button.
308      ; P1.1 = left mouse button.
309      ; P1.2 = right mouse button.
310      ;
311      ; 6-Feb-1991 This order of buttons reflects the current PCB layout
312      ; and is subject to change.
313      ;
314      $EJ
315      ; Locator messages (device defined)
316      ; send types
0003      LD_Position EQU 3 ; Device state report
317      ;
318      ; receive types
00B0      LD_Poll EQU 0B0h
0082      LD_SetInterval EQU 082h
319      ;
320      ;
321      ;
322      ;
323      ;
324      ; SelfTest errors (0=success)
0000      NO_ERROR EQU 0
0001      ROM_ERROR EQU 1
0002      RAM_ERROR EQU 2
325      ;
326      ;
327      ;
328      ;
329      ;
330      $EJ
331      ;*****
332      ; RAM usage *
333      ;*****
334      ; I2C variables
335      ; Register bank 0
336      ;
337      ;R0 - command parameter
338      ;R1 - index pointer
0002      I2CDat DATA 02h ;The byte being sent or received.
0003      BitCnt DATA 03h ;I2C bit counter
0004      ByteCnt DATA 04h ;I2C message byte counter
0005      ATNCnt DATA 05h ;ATN Retry counter
0006      I2CCxt DATA 06h ;I2C context, the event the CPU is waiting for.
0007      Temp DATA 07h ;All purpose temp
339      ;
340      ;
341      ;
342      ;
343      ;
344      ;
345      ; Desktop Bus Protocol
0008      MyAddr DATA 08h ;I2C address assigned this device.
0009      NACnt DATA 09h ;Negative Ack retry counter.
000A      RcvType DATA 0Ah ;Message or command type being received.
000B      SndType DATA 0Bh ;Message type being sent, or pending.
000C      MsgLen DATA 0Ch ;Message length field.
000D      Check DATA 0Dh ;Message checksum.
000E      RandH DATA 0Eh ;Random number (2 bytes)
000F      RandL DATA 0Fh
346      ;
347      ;
348      ;
349      ;
350      ;
351      ;
352      ;
353      ;
354      ;
355      ; Locator report buffer
0010      ReportBuf EQU 10h ;Beginning of position report buffer.
0010      Switch2 DATA 10h ;Switch data (Buttons 9 to 16)
0011      Switch1 DATA 11h ;Switch data (Buttons 1 to 8 )
0012      XBUF2 DATA 12h ;XData transmission buffer (MSB)
0013      XBUF1 DATA 13h ;XData transmission buffer (LSB)
0014      YBUF2 DATA 14h ;YData transmission buffer (MSB)

```

ACCESS.bus mouse application code for the
microcontroller

AN445

```

LOC OBJ      LINE      SOURCE
0015         362      YBUF1      DATA    15h      ;YData transmission buffer (LSB)
0016         363      ;Positive Y movement causes counter clockwise rotation of the encoder shaft.
0017         364      XCOUNT     DATA    16h      ;XData
0018         365      YCOUNT     DATA    17h      ;YData
0018         366      MARKER      EQU      YCOUNT+1
0018         367
0018         368      CapOffset    DATA    18h      ;Capabilities fragment offset
0019         369      CapLen       DATA    19h      ;Capabilities fragment length
001A         370      SelfTest     DATA    1Ah      ;Self Test result variable location
001B         371      RomSum      DATA    1Bh      ;Hold ROM checksum
001C         372      SampleClock DATA    1Ch      ;Time stamp of last sample
001C         373      ;3 spare.
001C         374
001C         375      ; Bit addressable area begins at 20h
001C         376      ; Location and switch compare variables
0020         377      LastXY       DATA    20h      ;Last X/Y Position (from P3)
0021         378      LastSW       DATA    21h      ;Last Switch Status (from P1)
0022         379      TranXY      DATA    22h      ;Port 3 transition register (bit addressable)
0022         380
0022         381      $EJ
0022         382      ; I2C status and position scanning flags.
0023         383      Flags        DATA    23h
0018         384      Prot         BIT      Flags.0 ;1=C/S message;0=device data stream.
0019         385      SendRpt      BIT      Flags.1 ;New Position Report flag.
001A         386      Movement    BIT      Flags.2 ;Movement detected flag.
001B         387      RxEnable    BIT      Flags.3 ;I2C receive enable.
001C         388      TxSelfRst    BIT      Flags.4 ;Indicates send Self-Reset after Assign (0).
001D         389      KeepID      BIT      Flags.5 ;Set means keep same device number.
001E         390      NotMyID     BIT      Flags.6
001F         391      AA          BIT      Flags.7 ;Assert Acknowledge.
001F         392
0024         393      FlagsA      DATA    24h
0020         394      MsgCheck     BIT      FlagsA.0 ;I2C message checksum or framing error.
0020         395
0020         396      ;11 spare
0030         397
0030         398      StackBase    DATA    30h ;16 byte stack
0030         399      ; need 2 bytes per subroutine
0030         400      ; 4 bytes per interrupt (max 2)
0030         401      ;RAM ends at 3Fh
0030         402      ; 50 bytes RAM used, 64 maximum.
0030         403
0030         404
0030         405      $EJ
0030         406      *****
0030         407      ; Code begins
0030         408      *****
0000         409      ORG 0000h ;Power up reset vector.
0000 01B4     410      AJMP PwrUp
0000         411
0003         412      ORG 003h ;INT0 is not used.
0003 01B4     413      AJMP PwrUp
0003         414
0003         415      ; Use this spare byte to hold the ROM checksum complement.
0005 51       416      DB 051h
0005         417
000B         418      ORG 00Bh ;Timer0 interrupt vector
000B 0125     419      AJMP Timer0 ;16 bit system tick generator
000B         420

```

ACCESS.bus mouse application code for the microcontroller

AN445

```

LOC  OBJ          LINE    SOURCE
0013          421        ORG      013h      ;INT1 is not used.
0013 01B4          422        AJMP    PwrUp
                                423
001B          424        ORG      01Bh      ;TimerI interrupt vector
001B 014F          425        AJMP    TimerI    ;I2C time out timer
                                426
0023          427        ORG      023h      ;I2C interrupt vector
0023 21D8          428        AJMP    I2CINT
                                429
                                430
                                431        $EJ
                                432        ;*****
                                433        ; Timer0 Interrupt
                                434        ; Position sampling interval time out.
                                435        ;*****
                                436
0025 C0D0          437        Timer0: PUSH    PSW
                                438        PUSH    ACC
                                439        MOV     A,MyAddr
                                440        CJNE    A,#Adr_Default,AddrOK
002B B46E02        441        SJMP     T0Exit
                                442
0030 E590          443        AddrOK: MOV     A,P1
                                444
                                445        CPL     A
                                446        ANL     A,#00000111b
0032 F4            447        CJNE    A,LastSW,CHNSWI
0033 5407          448        ;Switches did not change, check movement.
                                449        JNB     Movement,T0Exit
003B 8009          450        SJMP     T0Send
                                451
003D F521          452        CHNSWI: MOV     LastSW,A
                                453        ;Re-order switches from RLM to MRL until PCB is fixed.
                                454        RRC     A
                                455        MOV     ACC.2,C
                                456        ANL     A,#00000111b
0040 92E2          457        MOV     Switch1,A
                                458        ;Move to output buffer.
0042 5407          459        T0Send: SETB    SendRpt
                                460        SETB    I2CFG.MASTRQB
                                461
004A D0E0          462        T0Exit: POP     ACC
004C D0D0          463        POP     PSW
004E 32            464        RETI
                                465
                                466        $EJ
                                467        ;*****
                                468        ; TimerI interrupt
                                469        ; The I2C bus has timed out,
                                470        ; no SCL for at least 1020 machine cycles during an active frame.
                                471        ; Since SCL is stuck, we can't wait for DRDY.
                                472        ; Try to fix it manually.
                                473        ;*****
                                474
004F C2AF          475        TimerI: CLR     IE.EA
                                476        MOV     I2CFG,#CLRTI+CT
                                477        ; manually clear SLAVEN & MASTRQ.
0054 7598BC        478        MOV     I2CON,#CXA+CARL+CDR+CSTR+CSTP
0057 758130        479        MOV     SP,#StackBase
                                480

```

ACCESS.bus mouse application code for the microcontroller

AN445

```

LOC  OBJ          LINE  SOURCE
                                480
                                ; Attempt to regain control of the I2C bus after a bus fault.
005A  D280          482  FixBus:  SETB  SCL          ;Insure I/O port is not locking I2C.
005C  D281          483          SETB  SDA          ;If SCL is low, bus cannot be fixed.
005E  308020        484          JNB   SCL,ResetBus ;If SCL & SDA are high, force a stop.
0061  208113        485          JB    SDA,RStop    ;SDA is low, attempt to release SDA by clocking SCL.
                                486
0064  750309        487          MOV   BitCnt,#9      ;Set max # of tries to clear bus.
                                488  ClockBus: CLR   SCL          ;Force an I2C clock.
0067  C280          489          ACALL  SDelay
0069  11AF          490          JB    SDA,RStop    ;Did it work?
006B  208109        491          SETB  SCL
006E  D280          492          ACALL  SDelay
0070  11AF          493          DJNZ   BitCnt,ClockBus ;Repeat clocks until SDA clears or retry limit.
0072  D503F2        494          SJMP   ResetBus    ;Failed to fix bus by this method.
0075  800A          495
                                496  RStop: CLR   SDA          ;Try forcing a stop since
0077  C281          497          ACALL  SDelay    ; SCL & SDA are both high.
0079  11AF          498          SETB  SCL
007B  D280          499          ACALL  SDelay
007D  11AF          500          SETB  SDA
007F  D281          501          ;
                                502  ResetBus: ;Wait for bus to clear.
0081          503          JNB   SCL,$
0084  3081FD        504          JNB   SDA,$
0087  7401          505          MOV   A,#1        ;Pause for bus to stabilize.
0089  11A7          506          ACALL  LDelay
                                507          ;Re-enable I2C functions.
008B  750B00        508          MOV   SndType,#I_NoMsg    ;Cancel message if any.
008E  D21B          509          SETB  RxEnable    ;Enable receiving.
0090  1194          510          ACALL  InitI2C
                                511          ;Initialize I2C.
0092  2154          511          AJMP   MAIN      ;Restart MAIN.
                                512
                                513
                                514          ; Initialize I2C functions.
0094  75D893        515          InitI2C: MOV   I2CFG,#SLAVEN+TIRUN+CT ;Enable I2C
                                516          ;Set I2C to be idle receiver & clear all flags.
0097  7598FC        517          MOV   I2CON,#CXA+IDLE+CDR+CRL+CSTR+CSTP
009A  750601        518          MOV   I2CCxt,#RXIDLE    ;Context idle receiver
009D  31F3          519          ACALL  XRETI      ;Clear pending interrupt if any.
009F  75A89A        520          MOV   IE,#IntEnab    ;Enable interrupts (EA+EI2+ET1+ET0)
00A2  7410          521          MOV   A,#16
00A4  11A7          522          ACALL  LDelay    ;Wait to sync message frame.
00A6  22           523          RET
                                524
                                525          ; Long Delay, A/2 milliseconds.
00A7  7FA6          526          LDelay: MOV   R7,#166
00A9  DFFE          527          DJNZ   R7,$
00AB  D5E0F9        528          DJNZ   ACC,LDelay
00AE  22           529          RET
                                530
                                531          ; Short delay routine (10 machine cycles).
00AF  11B1          532          SDelay: ACALL  SD1
00B1  00           533          SD1:  NOP
00B2  00           534          NOP
00B3  22           535          RET
                                536
                                537          $EJ
                                538          ;*****

```

ACCESS.bus mouse application code for the microcontroller

AN445

```

LOC  OBJ          LINE  SOURCE
                                ; Power up initialization starts here
539                                ;*****
540                                ;*****
541                                ; Reset command branches to here.
00B4 75086E      542  PwrUp: MOV      MyAddr, #Adr_Default      ;Re-initialize default address.
00B7 E4          543          CLR      A
00B8 F5A8        544          MOV      IE, A ;Disable interrupts.
00BA 75D803      545          MOV      I2CFG, #CT ;Disable I2C
00BD 53D0E7      546          ANL      PSW, #0E7h ;Select RBO.
00C0 758130      547          MOV      SP, #StackBase
                                ; Initialize I/O pins
548
00C3 758003      549          MOV      P0, #03h ;Initialize I2C I/O pins, SCL & SDA high
00C6 7590FF      550          MOV      P1, #0FFh ;P1 set for input to read switches.
00C9 75B0FF      551          MOV      P3, #0FFh ;P3 set for input to read X-Y.
                                ; Initialize Timer0
552
00CC 758DDb      553          MOV      RTH, #DEF_RTH ;14 ms at 8 MHz
00CF 758BB8      554          MOV      RTL, #DEF_RTL
00D2 758810      555          MOV      TCON, #INIT_TCON ;Running, internal mode clock/12.
556
557                                ;*****
558                                ; Perform ROM Test (0-7Ffh, 2K bytes)
559                                ;*****
00D5 75F000      560  TestROM: MOV      B, #0 ;Initialize sum
00D8 900000      561          MOV      DPTR, #0000h ;Set pointer to start of ROM
00DB C3          562          CLR      C
00DC E4          563          SumLp: CLR      A
00DD 93          564          MOV      A, @A+DPTR ;Get byte from ROM
00DE 35F0        565          ADDC      A, B ;Add sum
00E0 F5F0        566          MOV      B, A ;Save sum in B
00E2 A3          567          INC      DPTR
00E3 E583        568          MOV      A, DPH ;Check if ROM complete
00E5 B408F4      569          CJNE      A, #08h, SumLp
00E8 5002        570          JNC      TestSum ;Add carry if set
00EA 05F0        571          INC      B
00EC E5F0        572  TestSum: MOV      A, B
00EE 6007        573          JZ      TestRAM ;If zero, ROM is Okay
00F0 751A01      574          MOV      SelfTest, #ROM_ERROR
00F3 F51B        575          MOV      RomSum, A ;Save bad checksum.
00F5 8023        576          SJMP      BadMem1
577
578          $EJ
579                                ;*****
580                                ; Perform RAM test (0-3Fh, 64 bytes)
581                                ; Does not test special function registers.
582                                ; A, B, and R0 are not preserved.
583                                ;*****
00F7 78AA        584  TestRAM: MOV      R0, #0AAh ;Test RAM location 0.
00F9 B8AA1B      585          CJNE      R0, #0AAh, BadMem
00FC 7855        586          MOV      R0, #055h
00FE B85516      587          CJNE      R0, #055h, BadMem
0101 783F        588          MOV      R0, #3Fh ;Init R0 to top of RAM.
0103 74AA        589          MOV      A, #0AAh ;Test alternate bits.
0105 86F0        590  ChkRAM: MOV      B, @R0 ;Save previous contents.
0107 F6          591          MOV      @R0, A
0108 B6AA0C      592          CJNE      @R0, #0AAh, BadMem
010B 23          593          RL      A ;Test other bits.
010C F6          594          MOV      @R0, A
010D B65507      595          CJNE      @R0, #055h, BadMem
0110 23          596          RL      A
0111 A6F0        597          MOV      @R0, B ;Restore contents.

```

ACCESS.bus mouse application code for the microcontroller **AN445**

LOC	OBJ	LINE	SOURCE
0113	D8F0	598	DJNZ R0, ChkRAM
0115	8005	599	SJMP MemOK
0117	751A02	600	BadMem: MOV SelfTest, #RAM_ERROR
		601	; Report bad memory. Since a memory problem was detected, the
		602	; normal I2C transmit code may be unreliable. Hope it isn't
		603	; a fatal problem and use it anyway. There's only so much
		604	; we can do. Could add special code here.
011A	8005	606	BadMem1: SJMP InitRAM
		607	
011C	751A00B	608	MemOK: MOV SelfTest, #0
011F	8000	609	SJMP InitRAM
		610	
		611	
		612	SEJ
		613	;*****
		614	;Initialize RAM
		615	;*****
		616	InitRAM: CLR A
0121	E4	617	MOV RcvType, A
0122	F50A	618	MOV SndType, A
0124	F50B	619	MOV R1, #ReportBuf
0126	7910	620	CLR Buf: MOV R1, A
0128	F7	621	INC R1
012A	B918FB	622	CJNE R1, #Marker, ClrBuf
012D	F523	623	MOV Flags, A
012F	F521	624	MOV LastSW, A
0131	750905	625	MOV NACnt, #5
		626	
		627	;Init LastXY
0134	E5B0	628	MOV A, P3
0136	540F	629	ANL A, #00Fh
0138	F520	630	MOV LastXY, A
		631	
		632	;*****
		633	;Set up to transmit self test report
		634	;*****
013A	C21B	635	CLR RxEnable
013C	1194	636	SetUp: ACALL InitI2C
		637	
013E	750601	638	Report: MOV I2CCxt, #RXIDLE
0141	750BE0	639	MOV SndType, #I_Attn
0144	D2DE	640	SETB I2CFG.MASTRQB
0146	20DEFD	641	JB I2CFG.MASTRQB, \$
0149	E51A	642	MOV A, SelfTest
014B	6005	643	MJZ RepDn
		644	
		645	;Selftest failed
		646	;Send 2nd report and try to start anyway.
014D	751A00	647	MOV SelfTest, #NO_ERROR
0150	80EC	648	SJMP Report
		649	
0152	D21B	650	RepDn: SETB RxEnable
		651	
		652	; fall through to MAIN.
		653	SEJ
		654	;*****
		655	; Main Routine
		656	

ACCESS.bus mouse application code for the microcontroller

AN445

```

LOC  OBJ          LINE      SOURCE
657          ; Sample X-Y quadrature inputs and compute mouse movement.
658          ; Accuracy requirement is: 0+/- 3% 0-10 inches per second.
659          ; 10 inches per second @ 200 dpi means up to 2000 input changes/second.
660          ; Minimum (Nyquist) sampling rate is 4000/sec or sample every 250µs.
661          ; This is only two character times at 80k bps.
662          ; For best accuracy, should sample between every I2C character.
663          ;
664          ; Sample timing with 8 MHz crystal:
665          ; no transition: 6 cycles 9µs
666          ; X or Y transition: 25-31 cycles 38-47µs
667          ; X and Y transition: 42-46 cycles 63-69µs
668          ;
669          ;*****
670          MAIN:
671          MOV      A,TL          ;Read timer to wait at least
672          SUBB     A,SampleClock; 64 cycles (96 µsec) between samples.
673          JNB     ACC.6,MAIN     ;
674          ;Take a sample
675          ;CLR      IE.EA         ;Protect sample code from interrupts.
676          MOV      SampleClock,TL
677          YACALL   Sample        ;Sample X-Y quadrature inputs.
678          ;SETB     IE.EA
679          SJMP     MAIN
680          ;
681          ; Sample X-Y quadrature inputs and update X-Y counters.
682          ; b3-b0 = YB YA XB XA.
683          ;
684          ; Channel A: 0 1 1 0 0 1 1 0 0 --->positive movement
685          ; Channel B: 0 0 1 1 0 0 1 1 0 <---negative movement
686          ;
687          ; A and C are not preserved.
688          ;
689          Sample: MOV      A,P3          ;Read X & Y position detectors.
690          ANL      A,#00001111B        ;We only need the low 4 bits
691          CJNE     A,LastXY,TRAN       ;Compare to last image.
692          RET                          ; If no change, return.
693          ;
694          TRAN: MOV      TranXY,LastXY  ;Set up to calculate XY transition.
695          MOV      LastXY,A            ;Save new P3 image.
696          XRL      TranXY,A            ;Mark transition bits (1=changed).
697          ;
698          XPULSE: JB      TranXY.0,XA   ;Branch on bit transitions.
699          JB      TranXY.1,XB
700          YPULSE: JB      TranXY.2,YA
701          JB      TranXY.3,YB
702          RET
703          ;
704          ;Decode direction from quadrature.
705          ;Change in XA
706          XA: MOV      A,LastXY
707          ANL      A,#00000011B        ;Get X'X
708          JZ       XDEC                ;If 00, backward
709          XRL      A,#00000011B
710          JZ       XDEC                ;If 11, backward
711          SJMP     XINC                ;01 or 10, forward.
712          ;Change in XB
713          XB: MOV      A,LastXY
714          ANL      A,#00000011B
715          JZ       XDEC                ;If 00, backward
716          XRL      A,#00000011B
717          JZ       XDEC                ;If 11, backward
718          SJMP     XINC                ;01 or 10, forward.
719          RET
720          ;
721          ;
722          ;
723          ;
724          ;
725          ;
726          ;
727          ;
728          ;
729          ;
730          ;
731          ;
732          ;
733          ;
734          ;
735          ;
736          ;
737          ;
738          ;
739          ;
740          ;
741          ;
742          ;
743          ;
744          ;
745          ;
746          ;
747          ;
748          ;
749          ;
750          ;
751          ;
752          ;
753          ;
754          ;
755          ;
756          ;
757          ;
758          ;
759          ;
760          ;
761          ;
762          ;
763          ;
764          ;
765          ;
766          ;
767          ;
768          ;
769          ;
770          ;
771          ;
772          ;
773          ;
774          ;
775          ;
776          ;
777          ;
778          ;
779          ;
780          ;
781          ;
782          ;
783          ;
784          ;
785          ;
786          ;
787          ;
788          ;
789          ;
790          ;
791          ;
792          ;
793          ;
794          ;
795          ;
796          ;
797          ;
798          ;
799          ;
800          ;
801          ;
802          ;
803          ;
804          ;
805          ;
806          ;
807          ;
808          ;
809          ;
810          ;
811          ;
812          ;
813          ;
814          ;
815          ;
816          ;
817          ;
818          ;
819          ;
820          ;
821          ;
822          ;
823          ;
824          ;
825          ;
826          ;
827          ;
828          ;
829          ;
830          ;
831          ;
832          ;
833          ;
834          ;
835          ;
836          ;
837          ;
838          ;
839          ;
840          ;
841          ;
842          ;
843          ;
844          ;
845          ;
846          ;
847          ;
848          ;
849          ;
850          ;
851          ;
852          ;
853          ;
854          ;
855          ;
856          ;
857          ;
858          ;
859          ;
860          ;
861          ;
862          ;
863          ;
864          ;
865          ;
866          ;
867          ;
868          ;
869          ;
870          ;
871          ;
872          ;
873          ;
874          ;
875          ;
876          ;
877          ;
878          ;
879          ;
880          ;
881          ;
882          ;
883          ;
884          ;
885          ;
886          ;
887          ;
888          ;
889          ;
890          ;
891          ;
892          ;
893          ;
894          ;
895          ;
896          ;
897          ;
898          ;
899          ;
900          ;
901          ;
902          ;
903          ;
904          ;
905          ;
906          ;
907          ;
908          ;
909          ;
910          ;
911          ;
912          ;
913          ;
914          ;
915          ;
916          ;
917          ;
918          ;
919          ;
920          ;
921          ;
922          ;
923          ;
924          ;
925          ;
926          ;
927          ;
928          ;
929          ;
930          ;
931          ;
932          ;
933          ;
934          ;
935          ;
936          ;
937          ;
938          ;
939          ;
940          ;
941          ;
942          ;
943          ;
944          ;
945          ;
946          ;
947          ;
948          ;
949          ;
950          ;
951          ;
952          ;
953          ;
954          ;
955          ;
956          ;
957          ;
958          ;
959          ;
960          ;
961          ;
962          ;
963          ;
964          ;
965          ;
966          ;
967          ;
968          ;
969          ;
970          ;
971          ;
972          ;
973          ;
974          ;
975          ;
976          ;
977          ;
978          ;
979          ;
980          ;
981          ;
982          ;
983          ;
984          ;
985          ;
986          ;
987          ;
988          ;
989          ;
990          ;
991          ;
992          ;
993          ;
994          ;
995          ;
996          ;
997          ;
998          ;
999          ;
1000         ;
1001         ;
1002         ;
1003         ;
1004         ;
1005         ;
1006         ;
1007         ;
1008         ;
1009         ;
1010         ;
1011         ;
1012         ;
1013         ;
1014         ;
1015         ;
1016         ;
1017         ;
1018         ;
1019         ;
1020         ;
1021         ;
1022         ;
1023         ;
1024         ;
1025         ;
1026         ;
1027         ;
1028         ;
1029         ;
1030         ;
1031         ;
1032         ;
1033         ;
1034         ;
1035         ;
1036         ;
1037         ;
1038         ;
1039         ;
1040         ;
1041         ;
1042         ;
1043         ;
1044         ;
1045         ;
1046         ;
1047         ;
1048         ;
1049         ;
1050         ;
1051         ;
1052         ;
1053         ;
1054         ;
1055         ;
1056         ;
1057         ;
1058         ;
1059         ;
1060         ;
1061         ;
1062         ;
1063         ;
1064         ;
1065         ;
1066         ;
1067         ;
1068         ;
1069         ;
1070         ;
1071         ;
1072         ;
1073         ;
1074         ;
1075         ;
1076         ;
1077         ;
1078         ;
1079         ;
1080         ;
1081         ;
1082         ;
1083         ;
1084         ;
1085         ;
1086         ;
1087         ;
1088         ;
1089         ;
1090         ;
1091         ;
1092         ;
1093         ;
1094         ;
1095         ;
1096         ;
1097         ;
1098         ;
1099         ;
1100         ;
1101         ;
1102         ;
1103         ;
1104         ;
1105         ;
1106         ;
1107         ;
1108         ;
1109         ;
1110         ;
1111         ;
1112         ;
1113         ;
1114         ;
1115         ;
1116         ;
1117         ;
1118         ;
1119         ;
1120         ;
1121         ;
1122         ;
1123         ;
1124         ;
1125         ;
1126         ;
1127         ;
1128         ;
1129         ;
1130         ;
1131         ;
1132         ;
1133         ;
1134         ;
1135         ;
1136         ;
1137         ;
1138         ;
1139         ;
1140         ;
1141         ;
1142         ;
1143         ;
1144         ;
1145         ;
1146         ;
1147         ;
1148         ;
1149         ;
1150         ;
1151         ;
1152         ;
1153         ;
1154         ;
1155         ;
1156         ;
1157         ;
1158         ;
1159         ;
1160         ;
1161         ;
1162         ;
1163         ;
1164         ;
1165         ;
1166         ;
1167         ;
1168         ;
1169         ;
1170         ;
1171         ;
1172         ;
1173         ;
1174         ;
1175         ;
1176         ;
1177         ;
1178         ;
1179         ;
1180         ;
1181         ;
1182         ;
1183         ;
1184         ;
1185         ;
1186         ;
1187         ;
1188         ;
1189         ;
1190         ;
1191         ;
1192         ;
1193         ;
1194         ;
1195         ;
1196         ;
1197         ;
1198         ;
1199         ;
1200         ;
1201         ;
1202         ;
1203         ;
1204         ;
1205         ;
1206         ;
1207         ;
1208         ;
1209         ;
1210         ;
1211         ;
1212         ;
1213         ;
1214         ;
1215         ;
1216         ;
1217         ;
1218         ;
1219         ;
1220         ;
1221         ;
1222         ;
1223         ;
1224         ;
1225         ;
1226         ;
1227         ;
1228         ;
1229         ;
1230         ;
1231         ;
1232         ;
1233         ;
1234         ;
1235         ;
1236         ;
1237         ;
1238         ;
1239         ;
1240         ;
1241         ;
1242         ;
1243         ;
1244         ;
1245         ;
1246         ;
1247         ;
1248         ;
1249         ;
1250         ;
1251         ;
1252         ;
1253         ;
1254         ;
1255         ;
1256         ;
1257         ;
1258         ;
1259         ;
1260         ;
1261         ;
1262         ;
1263         ;
1264         ;
1265         ;
1266         ;
1267         ;
1268         ;
1269         ;
1270         ;
1271         ;
1272         ;
1273         ;
1274         ;
1275         ;
1276         ;
1277         ;
1278         ;
1279         ;
1280         ;
1281         ;
1282         ;
1283         ;
1284         ;
1285         ;
1286         ;
1287         ;
1288         ;
1289         ;
1290         ;
1291         ;
1292         ;
1293         ;
1294         ;
1295         ;
1296         ;
1297         ;
1298         ;
1299         ;
1300         ;
1301         ;
1302         ;
1303         ;
1304         ;
1305         ;
1306         ;
1307         ;
1308         ;
1309         ;
1310         ;
1311         ;
1312         ;
1313         ;
1314         ;
1315         ;
1316         ;
1317         ;
1318         ;
1319         ;
1320         ;
1321         ;
1322         ;
1323         ;
1324         ;
1325         ;
1326         ;
1327         ;
1328         ;
1329         ;
1330         ;
1331         ;
1332         ;
1333         ;
1334         ;
1335         ;
1336         ;
1337         ;
1338         ;
1339         ;
1340         ;
1341         ;
1342         ;
1343         ;
1344         ;
1345         ;
1346         ;
1347         ;
1348         ;
1349         ;
1350         ;
1351         ;
1352         ;
1353         ;
1354         ;
1355         ;
1356         ;
1357         ;
1358         ;
1359         ;
1360         ;
1361         ;
1362         ;
1363         ;
1364         ;
1365         ;
1366         ;
1367         ;
1368         ;
1369         ;
1370         ;
1371         ;
1372         ;
1373         ;
1374         ;
1375         ;
1376         ;
1377         ;
1378         ;
1379         ;
1380         ;
1381         ;
1382         ;
1383         ;
1384         ;
1385         ;
1386         ;
1387         ;
1388         ;
1389         ;
1390         ;
1391         ;
1392         ;
1393         ;
1394         ;
1395         ;
1396         ;
1397         ;
1398         ;
1399         ;
1400         ;
1401         ;
1402         ;
1403         ;
1404         ;
1405         ;
1406         ;
1407         ;
1408         ;
1409         ;
1410         ;
1411         ;
1412         ;
1413         ;
1414         ;
1415         ;
1416         ;
1417         ;
1418         ;
1419         ;
1420         ;
1421         ;
1422         ;
1423         ;
1424         ;
1425         ;
1426         ;
1427         ;
1428         ;
1429         ;
1430         ;
1431         ;
1432         ;
1433         ;
1434         ;
1435         ;
1436         ;
1437         ;
1438         ;
1439         ;
1440         ;
1441         ;
1442         ;
1443         ;
1444         ;
1445         ;
1446         ;
1447         ;
1448         ;
1449         ;
1450         ;
1451         ;
1452         ;
1453         ;
1454         ;
1455         ;
1456         ;
1457         ;
1458         ;
1459         ;
1460         ;
1461         ;
1462         ;
1463         ;
1464         ;
1465         ;
1466         ;
1467         ;
1468         ;
1469         ;
1470         ;
1471         ;
1472         ;
1473         ;
1474         ;
1475         ;
1476         ;
1477         ;
1478         ;
1479         ;
1480         ;
1481         ;
1482         ;
1483         ;
1484         ;
1485         ;
1486         ;
1487         ;
1488         ;
1489         ;
1490         ;
1491         ;
1492         ;
1493         ;
1494         ;
1495         ;
1496         ;
1497         ;
1498         ;
1499         ;
1500         ;
1501         ;
1502         ;
1503         ;
1504         ;
1505         ;
1506         ;
1507         ;
1508         ;
1509         ;
1510         ;
1511         ;
1512         ;
1513         ;
1514         ;
1515         ;
1516         ;
1517         ;
1518         ;
1519         ;
1520         ;
1521         ;
1522         ;
1523         ;
1524         ;
1525         ;
1526         ;
1527         ;
1528         ;
1529         ;
1530         ;
1531         ;
1532         ;
1533         ;
1534         ;
1535         ;
1536         ;
1537         ;
1538         ;
1539         ;
1540         ;
1541         ;
1542         ;
1543         ;
1544         ;
1545         ;
1546         ;
1547         ;
1548         ;
1549         ;
1550         ;
1551         ;
1552         ;
1553         ;
1554         ;
1555         ;
1556         ;
1557         ;
1558         ;
1559         ;
1560         ;
1561         ;
1562         ;
1563         ;
1564         ;
1565         ;
1566         ;
1567         ;
1568         ;
1569         ;
1570         ;
1571         ;
1572         ;
1573         ;
1574         ;
1575         ;
1576         ;
1577         ;
1578         ;
1579         ;
1580         ;
1581         ;
1582         ;
1583         ;
1584         ;
1585         ;
1586         ;
1587         ;
1588         ;
1589         ;
1590         ;
1591         ;
1592         ;
1593         ;
1594         ;
1595         ;
1596         ;
1597         ;
1598         ;
1599         ;
1600         ;
1601         ;
1602         ;
1603         ;
1604         ;
1605         ;
1606         ;
1607         ;
1608         ;
1609         ;
1610         ;
1611         ;
1612         ;
1613         ;
1614         ;
1615         ;
1616         ;
1617         ;
1618         ;
1619         ;
1620         ;
1621         ;
1622         ;
1623         ;
1624         ;
1625         ;
1626         ;
1627         ;
1628         ;
1629         ;
1630         ;
1631         ;
1632         ;
1633         ;
1634         ;
1635         ;
1636         ;
1637         ;
1638         ;
1639         ;
1640         ;
1641         ;
1642         ;
1643         ;
1644         ;
1645         ;
1646         ;
1647         ;
1648         ;
1649         ;
1650         ;
1651         ;
1652         ;
1653         ;
1654         ;
1655         ;
1656         ;
1657         ;
1658         ;
1659         ;
1660         ;
1661         ;
1662         ;
1663         ;
1664         ;
1665         ;
1666         ;
1667         ;
1668         ;
1669         ;
1670         ;
1671         ;
1672         ;
1673         ;
1674         ;
1675         ;
1676         ;
1677         ;
1678         ;
1679         ;
1680         ;
1681         ;
1682         ;
1683         ;
1684         ;
1685         ;
1686         ;
1687         ;
1688         ;
1689         ;
1690         ;
1691         ;
1692         ;
1693         ;
1694         ;
1695         ;
1696         ;
1697         ;
1698         ;
1699         ;
1700         ;
1701         ;
1702         ;
1703         ;
1704         ;
1705         ;
1706         ;
1707         ;
1708         ;
1709         ;
1710         ;
1711         ;
1712         ;
1713         ;
1714         ;
1715         ;
1716         ;
1717         ;
1718         ;
1719         ;
1720         ;
1721         ;
1722         ;
1723         ;
1724         ;
1725         ;
1726         ;
1727         ;
1728         ;
1729         ;
1730         ;
1731         ;
1732         ;
1733         ;
1734         ;
1735         ;
1736         ;
1737         ;
1738         ;
1739         ;
1740         ;
1741         ;
1742         ;
1743         ;
1744         ;
1745         ;
1746         ;
1747         ;
1748         ;
1749         ;
1750         ;
1751         ;
1752         ;
1753         ;
1754         ;
1755         ;
1756         ;
1757         ;
1758         ;
1759         ;
1760         ;
1761         ;
1762         ;
1763         ;
1764         ;
1765         ;
1766         ;
1767         ;
1768         ;
1769         ;
1770         ;
1771         ;
1772         ;
1773         ;
1774         ;
1775         ;
1776         ;
1777         ;
1778         ;
1779         ;
1780         ;
1781         ;
1782         ;
1783         ;
1784         ;
1785         ;
1786         ;
1787         ;
1788         ;
1789         ;
1790         ;
1791         ;
1792         ;
1793         ;
1794         ;
1795         ;
1796         ;
1797         ;
1798         ;
1799         ;
1800         ;
1801         ;
1802         ;
1803         ;
1804         ;
1805         ;
1806         ;
1807         ;
1808         ;
1809         ;
1810         ;
1811         ;
1812         ;
1813         ;
1814         ;
1815         ;
1816         ;
1817         ;
1818         ;
1819         ;
1820         ;
1821         ;
1822         ;
1823         ;
1824         ;
1825         ;
1826         ;
1827         ;
1828         ;
1829         ;
1830         ;
1831         ;
1832         ;
1833         ;
1834         ;
1835         ;
1836         ;
1837         ;
1838         ;
1839         ;
1840         ;
1841         ;
1842         ;
1843         ;
1844         ;
1845         ;
1846         ;
1847         ;
1848         ;
1849         ;
1850         ;
1851         ;
1852         ;
1853         ;
1854         ;
1855         ;
1856         ;
1857         ;
1858         ;
1859         ;
1860         ;
1861         ;
1862         ;
1863         ;
1864         ;
1865         ;
1866         ;
1867         ;
1868         ;
1869         ;
1870         ;
1871         ;
1872         ;
1873         ;
1874         ;
1875         ;
1876         ;
1877         ;
1878         ;
1879         ;
1880         ;
1881         ;
1882         ;
1883         ;
1884         ;
1885         ;
1886         ;
1887         ;
1888         ;
1889         ;
1890         ;
1891         ;
1892         ;
1893         ;
1894         ;
1895         ;
1896         ;
1897         ;
1898         ;
1899         ;
1900         ;
1901         ;
1902         ;
1903         ;
1904         ;
1905         ;
1906         ;
1907         ;
1908         ;
1909         ;
1910         ;
1911         ;
1912         ;
1913         ;
1914         ;
1915         ;
1916         ;
1917         ;
1918         ;
1919         ;
1920         ;
1921         ;
1922         ;
1923         ;
1924         ;
1925         ;
1926         ;
1927         ;
1928         ;
1929         ;
1930         ;
1931         ;
1932         ;
1933         ;
1934         ;
1935         ;
1936         ;
1937         ;
1938         ;
1939         ;
1940         ;
1941         ;
1942         ;
1943         ;
1944         ;
1945         ;
1946         ;
1947         ;
1948         ;
1949         ;
1950         ;
1951         ;
1952         ;
1953         ;
1954         ;
1955         ;
1956         ;
1957         ;
1958         ;
1959         ;
1960         ;
1961         ;
1962         ;
1963         ;
1964         ;
1965         ;
1966         ;
1967         ;
1968         ;
1969         ;
1970         ;
1971         ;
1972         ;
1973         ;
1974         ;
1975         ;
1976         ;
1977         ;
1978         ;
1979         ;
1980         ;
1981         ;
1982         ;
1983         ;
1984         ;
1985         ;
1986         ;
1987         ;
1988         ;
1989         ;
1990         ;
1991         ;
1992         ;
1993         ;
1994         ;
1995         ;
1996         ;
1997         ;
1998         ;
1999         ;
2000         ;
2001         ;
2002         ;
2003         ;
2004         ;
2005         ;
2006         ;
2007         ;
2008         ;
2009         ;
2010         ;
2011         ;
2012         ;
2013         ;
2014         ;
2015         ;
2016         ;
2017         ;
2018         ;
2019         ;
2020         ;
2021         ;
2022         ;
2023         ;
2024         ;
2025         ;
2026         ;
2027         ;
2028         ;
2029         ;
2030         ;
2031         ;
2032         ;
2033         ;
2034         ;
2035         ;
2036         ;
2037         ;
2038         ;
2039         ;
2040         ;
2041         ;
2042         ;
2043         ;
2044         ;
2045         ;
2046         ;
2047         ;
2048         ;
2049         ;
2050         ;
2051         ;
2052         ;
2053         ;
2054         ;
2055         ;
2056         ;
2057         ;
2058         ;
2059         ;
2060         ;
2061         ;
2062         ;
2063         ;
2064         ;
2065         ;
2066         ;
2067         ;
2068         ;
2069         ;
2070         ;
2071         ;
2072         ;
2073         ;
2074         ;
2075         ;
2076         ;
2077         ;
2078         ;
2079         ;
2080         ;
2081         ;
2082         ;
2083         ;
2084         ;
2085         ;
2086         ;
2087         ;
2088         ;
2089         ;
2090         ;
2091         ;
2092         ;
2093         ;
2094         ;
2095         ;
2096         ;
2097         ;
2098         ;
2099         ;
2100         ;
2101         ;
2102         ;
2103         ;
2104         ;
2105         ;
2106         ;
2107         ;
2108         ;
2109         ;
2110         ;
2111         ;
2112         ;
2113         ;
2114         ;
2115         ;
2116         ;
2117         ;
2118         ;
2119         ;
2120         ;
2121         ;
2122         ;
2123         ;
2124         ;
2125         ;
2126         ;
2127         ;
2128         ;
2129         ;
2130         ;
2131         ;
2132         ;
2133         ;
2134         ;
2135         ;
2136         ;
2137         ;
2138         ;
2139         ;
2140         ;
2141         ;
2142         ;
2143         ;
2144         ;
2145         ;
2146         ;
2147         ;
2148         ;
2149         ;
2150         ;
2151         ;
2152         ;
2153         ;
2154         ;
2155         ;
2156         ;
2157         ;
2158         ;
2159         ;
2160         ;
2161         ;
2162         ;
2163         ;
2164         ;
2165         ;
2166         ;
2167         ;
2168         ;
2169         ;
2170         ;
2171         ;
2172         ;
2173         ;
2174         ;
2175         ;
2176         ;
2177         ;
2178         ;
2179         ;
2180         ;
2181         ;
2182         ;
2183         ;
2184         ;
2185         ;
2186         ;
2187         ;
2188         ;
2189         ;
2190         ;
2191         ;
2192         ;
2193         ;
2194         ;
2195         ;
2196         ;
2197         ;
2198         ;
2199         ;
2200         ;
2201         ;
2202         ;
2203         ;
2204         ;
2205         ;
2206         ;
2207         ;
2208         ;
2209         ;
2210         ;
2211         ;
2212         ;
2213         ;
2214         ;
2215         ;
2216         ;
2217         ;
2218         ;
2219         ;
2220         ;
2221         ;
2222         ;
2223         ;
2224         ;
2225         ;
2226         ;
2227         ;
2228         ;
2229         ;
2230         ;
2231         ;
2232         ;
2233         ;
2234         ;
2235         ;
2236         ;
2237         ;
2238         ;
2239         ;
2240         ;
2241         ;
2242         ;
2243         ;
2244         ;
2245         ;
2246         ;
2247         ;
2248         ;
2249         ;
2250         ;
2251         ;
2252         ;
2253         ;
2254         ;
2255         ;
2256         ;
2257         ;
2258         ;
2259         ;
2260         ;
2261         ;
2262         ;
2263         ;
2264         ;
2265         ;
2266         ;
2267         ;
2268         ;
2269         ;
2270         ;
2271         ;
2272         ;
2273         ;
2274         ;
2275         ;
2276         ;
2277         ;
2278         ;
2279         ;
2280         ;
2281         ;
2282         ;
2283         ;
2284         ;
2285         ;
2286         ;
2287         ;
2288         ;
2289         ;
2290         ;
2291         ;
2292         ;
2293         ;
2294         ;
2295         ;
2296         ;
2297         ;
2298         ;
2299         ;
2300         ;
2301         ;
2302         ;
2303         ;
2304         ;
2305         ;
2306         ;
2307         ;
2308         ;
2309         ;
2310         ;
2311         ;
2312         ;
2313         ;
2314         ;
2315         ;
2316         ;
2317         ;
2318         ;
2319         ;
2320         ;
2321         ;
2322         ;
2323         ;
2324         ;
2325         ;
2326         ;
2327         ;
2328         ;
2329         ;
2330         ;
2331         ;
2332         ;
2333         ;
2334         ;
2335         ;
2336         ;
2337         ;
2338         ;
2339         ;
2340         ;
2341         ;
2342         ;
2343         ;
2344         ;
2345         ;
2346         ;
2347         ;
2348         ;
2349         ;
2350         ;
2351         ;
2352         ;
2353         ;
2354         ;
23
```

ACCESS.bus mouse application code for the microcontroller

AN445

```

018C 5403      715      ANL      A,#00000011B      ;Get X' X
018E 6010      716      JZ       XINC      ;If 00, forward
0190 6403      717      XRL      A,#00000011B
0192 600C      718      JZ       XINC      ;If 11, forward
                        719      ;01 or 10, backward, fall through to decrement
                        720
0194 7480      721      XDEC:    MOV      A,#080h      ;Do not decrement if
0196 6516      722      XRL      A,XCOUNT      ; count already at minimum -127.
0198 6004      723      JZ       XDEC2
019A 1516      724      DEC      XCOUNT
019C D21A      725      SETB     Movement      ;Note position has changed.
019E 80D7      726      XDEC2:   SJMP     YPULSE      ;Check for possible Y pulse.
                        727
01A0 747F      728      XINC:    MOV      A,#07Fh      ;Do not increment if
01A2 6516      729      XRL      A,XCOUNT      ; count already at maximum 127.
01A4 6004      730      JZ       XINC2
01A6 0516      731      INC      XCOUNT
01A8 D21A      732      SETB     Movement      ;Note position has changed.
01AA 80CB      733      XINC2:   SJMP     YPULSE      ;Check for possible Y pulse.
                        734
                        735      ;Change in YA
01AC E520      736      YA:      MOV      A,LastXY
01AE 540C      737      ANL      A,#00001100B      ;Get Y' Y
01B0 6010      738      JZ       YDEC      ;If 00, backward
01B2 640C      739      XRL      A,#00001100B
01B4 600C      740      JZ       YDEC      ;If 11, backward
01B6 8015      741      SJMP     YINC      ;01 or 10, forward.
                        742      ;Change in YB
01B8 E520      743      YB:      MOV      A,LastXY
01BA 540C      744      ANL      A,#00001100B      ;Get Y' Y
01BC 600F      745      JZ       YINC      ;If 00, forward
01BE 640C      746      XRL      A,#00001100B
01C0 600B      747      JZ       YINC      ;If 11, forward
                        748      ;01 or 10, backward, fall through to decrement
                        749
01C2 7480      750      YDEC:    MOV      A,#080h      ;Do not decrement if
01C4 6517      751      XRL      A,YCOUNT      ; count already at minimum -127.
01C6 6004      752      JZ       YDEC2
01C8 1517      753      DEC      YCOUNT
01CA D21A      754      SETB     Movement      ;Note position has changed.
01CC 22        755      YDEC2:   RET
                        756
01CD 747F      757      YINC:    MOV      A,#07Fh      ;Do not increment if
01CF 6517      758      XRL      A,YCOUNT      ; count already at maximum 127.
01D1 6004      759      JZ       YINC2
01D3 0517      760      INC      YCOUNT
01D5 D21A      761      SETB     Movement      ;Note position has changed.
01D7 22        762      YINC2:   RET
                        763
                        764      $EJ
                        765      ; I2C message processing contexts:
0001          766      RXIDLE   EQU      1      ;Idle receiver waiting for start.
0002          767      RXBIT     EQU      2      ;Waiting to receive a bit.
0003          768      RXACK     EQU      3      ;Waiting for ACK to complete.
                        769
0004          770      TXBIT     EQU      4      ;Waiting to send a bit.
0005          771      TXREAD    EQU      5      ;Waiting to read ACK.
0006          772      TXACK     EQU      6      ;Waiting for ACK.
                        773
LOC OBJ      LINE      SOURCE

```

ACCESS.bus mouse application code for the microcontroller

AN445

	774				
	775				
	776	; I2C Interrupt Reasons			
	777	; Events CPU might be waiting for			
	778	; Receive			
	779	(1) Start signal detected by idle slave (DRDY)			
	780	; (2) Next bit received (DRDY)			
	781	; (3) Acknowledge has been sent (DRDY)			
	782	; Transmit			
	783	; (4) Bus mastership granted (START and MASTER)			
	784	; (5) Ready to transmit next bit (DRDY)			
	785	; (6) Acknowledge received (DRDY)			
	786	; Unsolicited			
	787	; (7) Arbitration loss (ARL)			
	788	; (8) Sender aborted message (STOP)			
	789	; (9) Sender started new message before slave became idle (START)			
	790				
	791	; Only some of these events can occur at any time depending on			
	792	; the state of sending or receiving a message.			
	793				
	794	; When the interrupt occurs, the keyboard needs to recover			
	795	; the context from which it was sending or receiving a			
	796	; message. This context is maintained as follows:			
	797				
	798	; I2Ccxt I2C context, what event is expected.			
	799	; I2CDat The byte being sent or received.			
	800	; BitCnt Where we are in the sending or receiving the byte.			
	801	; ByteCnt Where we are in sending or receiving a message			
	802	; Check Computed checksum.			
	803	; RcvType The message or command type being received.			
	804	; This may determine how successive bytes			
	805	; are to be processed.			
	806	; SndType Type of message being sent or pending.			
	807	; This will determine how bytes are transmitted.			
	808	; SendRpt Flag indicating CPU is waiting to send a Position			
	809	; report (and requested to become master).			
	810				
	811				
	812				
	813	; \$EJ			
	814	;*****			
	815	; Enter I2C Interrupt			
	816	;*****			
01D8 C2AC	817	I2CINT: CLR IE.EI2 ;disable the I2C interrupt			
01DA 31F3	818	ACALL XRETI ;then re-enable others			
01DC COD0	819	PUSH PSW ;save registers			
01DE C0E0	820	PUSH ACC			
	821				
	822	; Dispatch interrupt			
01E0 309911	823	DISPAT: JNB I2CON.MST,SLAVE			
01E3 41B1	824	AJMP MASTER ;go if we're master			
	825				
	826	; Wait for ATN			
01E5 7D08	827	WaitATN: MOV R5,#DelayATN ;Load ATN count (about 50mms)			
01E7 209EF6	828	Wait1: JB I2CON.ATN,DISPAT ;If ATN, dispatch next event,			
01EA DDFB	829	DJNZ R5,Wait1 ; else loop to try again.			
	830	; If not seen after count tries,			
	831	; return from I2C interrupt.			
	832				
LOC OBJ	LINE	SOURCE			

ACCESS.bus mouse application code for the microcontroller

AN445

```

833      ; Exit I2C interrupt
834      ; restore registers and return from interrupt
01EC D0E0      835      I2CRTI: POP      ACC      ; I2C interrupt
01EE D0D0      836      POP      PSW      ; Events CPU might be waiting for
01F0 D2AC      837      SETB      IE.EI2      ;re-enable I2C interrupt
01F2 22        838      RET      ;return to interrupted process
839
01F3 32        840      XRETI: RETI      ;used at start of service routine
841
842
843      $EJ
844      ;*****
845      ; SLAVE RECEIVER
846      ; R2=I2CDat, R3=BitCnt, R4=ByteCnt, R5=ATNCnt, R6=I2CCxt
847      ;*****
848      ; Handle DRDY
849      ;*****
01F4 309D5A    850      SLAVE: JNB      I2CON.DRDY,NDRDY ;Is it DRDY?
851      ; -context waiting for bit
01F7 BE0235    852      CJNE      R6,#RXBIT,NRxBit ;Context waiting for bit?
01FA EA        853      Rx0: MOV      A,R2 ;Yes, get data in A
01FB DB21      854      Rx1: DJNZ     R3,N8Bit ;8th bit?
855
856      ; Read 8th bit
01FD A29F      857      MOV      C,I2CON.RDAT ;Get 8th bit, don't clear ATN
01FF 33        858      RLC      A ;Include the 8th bit
0200 FA        859      MOV      R2,A ;Put data in R2.
0201 620D      860      XRL      Check,A ;XOR it to check
861
862      ;Send Acknowledge as appropriate.
0203 BC0007    863      CJNE      R4,#0,DoAck1 ;Address byte? (ByteCnt=0)
0206 B5080E    864      CJNE      A,MyAddr,NotMe ;Is it my address?
0209 F50D      865      MOV      Check,A ;Yes, initialize check
020B C21F      866      CLR      AA ;AA=Flags.7
020D 852399    867      DoAck1: MOV      I2DAT,Flags ;Assert Acknowledge (AA=Flags.7)
0210 7E03      868      MOV      R6,#RXACK ;Set context waiting for ACK
0212 209D1D    869      JB      I2CON.DRDY,AckCmp ;Can we skip waiting?
0215 21E5      870      AJMP     WaitATN ;Wait for ATN.
871      ; Not addressed to me
872      NotMe: MOV      R6,#RXIDLE ;Set context to be idle receiver
0217 7E01      873      MOV      I2CON,#CDR+CSTR+CSTP+CRL+IDLE
0219 75987C    874      AJMP     I2CRTI ;Resume interrupted activity
875
876      ;*****
877      ; Read bits 1-7
021E C2E7      877      N8Bit: CLR      ACC.7
0220 4599      878      ORL      A,I2DAT ;Include the bit, clear ATN.
0222 23        879      RL      A ;Data comes in at MSB.
0223 209DD5    880      JB      I2CON.DRDY,Rx1 ;If DRDY, short cut.
0226 209DD2    881      JB      I2CON.DRDY,Rx1
0229 209DCF    882      JB      I2CON.DRDY,Rx1 ;One more try.
022C FA        883      MOV      R2,A ;Put data back.
022D 21E5      884      AJMP     WaitATN
885
886      ; -context waiting for ACK to complete
022F BE0311    887      NRxBit: CJNE      R6,#RXACK,NRxAck ;Context waiting for ACK?
0232 7598A0    888      AckCmp: MOV      I2CON,#CDR+CXA ;ACK complete, clr xmt.
889
890      ;Process complete byte.
0235 618F      890      AJMP     DORXB
891
892      ; Return is AJMP DNRXB.
LOC OBJ      LINE      SOURCE

```

ACCESS.bus mouse application code for the microcontroller

AN445

```

0237 0C          892      DNRXB: INC      R4          ;Increment ByteCnt.
                   893      ;ACALL Sample
0238 7E02      894      SetRXB: MOV      R6,#RXBIT      ;Set context for next byte.
023A 7A00      895      MOV      R2,#0          ;Clear receive buffer
023C 7B08      896      MOV      R3,#8          ;BitCnt=8
023E 209DB9    897      JB      I2CON.DRDY,Rx0      ;If DRDY, short cut.
0241 21E5      898      AJMP     WaitATN      ;Wait for ATN
                   899
                   900      ; - context idle slave
0243 BE010B    901      NRxAck: CJNE     R6,#RXIDLE,NRxIdle ;Context idle slave?
0246 301BCE    902      JNB      RxEnable,NotMe      ;Am I enabled?
                   903      ;Yes, initialize to receive first byte
0249 7B07      904      MOV      R3,#7          ;BitCnt=7 (remaining)
024B E4        905      CLR      A              ;Data in A
024C FC        906      MOV      R4,A          ;ByteCnt=0
024D 7E02      907      MOV      R6,#RXBIT      ;I2CCxt=receive next bit
024F 411E      908      AJMP     N8Bit
                   909      ; Context was not waiting for Next bit, ACK, or Idle slave.
                   910      ; Could be ARL. Dispatch other flags.
                   911      ; Do not clear DRDY, we'll come back after ARL if necessary.
0251          912      NRxIdle:
                   913
                   914      ; Yes, clear
                   915      $EJ:
                   916      ;*****
                   917      ; It wasn't DRDY, could be ARL, START, or STOP
                   918      ; Handle ARL
                   919      ;*****
0251 309C38    920      NDRDY: JNB      I2CON.ARL,RxStop      ;Is it ARL?
                   921      ;Yes, note MASTRQ is still on unless we were sending STOP.
                   922      ;SndType is still pending. If it was a position report,
                   923      ;indicate pending report in SendRpt in case SndType is needed.
0254 30DE2E    924      ARL0: JNB      I2CFG.MASTRQB,Naddr ;Was I sending STOP?
0257 E50B      925      MOV      A,SndType
0259 B40305    926      CJNE     A,#LD_Position,ARL01      ;Was I sending position?
025C 750B00    927      MOV      SndType,#I_NoMsg
025F D219      928      SETB     SendRpt
0261 B4E405    929      ARL01: CJNE     A,#I_Error,ARL1      ;Was I sending Error?
0264 750B00    930      MOV      SndType,#I_NoMsg
0267 D220      931      SETB     MsgCheck
0269 301B19    932      ARL1: JNB      RxEnable,NAddr      ;Am I enabled?
026C 209B39    933      JB      I2CON.STR,SlvStart      ;Handle start.
026F BC000A    934      CJNE     R4,#0,ARL3          ;Did we ARL in Address (ByteCnt=0)?
                   935      ; Lost arbitration in address, set context to read rest
                   936      ; of address in case message is to me.
0272 759810    937      ARL2: MOV      I2CON,#CARL      ;Clear ARL.
0275 7E02      938      MOV      R6,#RXBIT      ;Set context waiting to read bit.
0277 209D80    939      JB      I2CON.DRDY,Rx0      ;If DRDY, short cut.
027A 21E5      940      AJMP     WaitATN
                   941      ; Lost arbitration outside dst addr
027C B4F006    942      ARL3: CJNE     A,#I_Reset,NAddr      ;Was I sending I_Reset to my Addr?
027F C21C      943      CLR      TxSelfRst      ;Yes, reset flag since I lost.
0281 C21F      944      CLR      AA          ;Acknowledge received bytes.
0283 80ED      945      SJMP     ARL2          ;Message must be for me.
                   946      ; Message not for me, go back to idle receive.
0285          947      NAddr:
0285 7598FC    948      Idles: MOV      I2CON,#CXA+IDLE+CARL+CDR+CSTR+CSTP
0288 7E01      949      MOV      R6,#RXIDLE
028A 21EC      950      AJMP     I2CRTI
LOC OBJ      LINE      SOURCE

```


ACCESS.bus mouse application code for the
microcontroller

AN445

```

951 $EJ
952 ;*****
953 ; Handle STOP
954 ;*****
028C 309A16 955 RxStop: JNB I2CON.STP,RxStart ;Confirm it was Stop.
028F 759804 956 MOV I2CON,#CSTP ;Clear it.
0292 201F04 957 JB AA,RxStop0 ;Check end of message reached,
958 ; Assert Acknowledge=1.
959 ; Received STOP before end of message.
0295 D220 960 SETB MsgCheck ;Signal interface error.
0297 D2DE 961 SETB I2CFG.MASTRQB
0299 7E01 962 RxStop0: MOV R6,#RXIDLE ;Set context idle receiver.
029B 309E02 963 JNB I2CON.ATN,RxStop1 ;If ATN, dispatch next event
029E 21E0 964 AJMP DISPAT
02A0 759840 965 RxStop1: MOV I2CON,#IDLE ;Become idle.
02A3 21EC 966 AJMP I2CRTI ;Resume interrupted activity
967
968 ;*****
969 ; Handle START
970 ;*****
02A5 309B07 971 RxStart: JNB I2CON.STR,RxFault ;Was it start?
02A8 972 SlvStart:
02A8 759818 973 MOV I2CON,#CSTR+CARL ;Yes, clear it.
02AB 7C00 974 MOV R4,#0 ;ByteCnt=0
02AD 4138 975 AJMP SetRXB ;Set up to receive byte.
976
977 ; It wasn't DRDY, ARL, START, or STOP. Inconsistency error.
02AF 01B4 978 RxFault: AJMP PwrUp
979
980
981 $EJ
982 ;*****
983 ; MASTER TRANSMITTER
984 ; R2=I2CDat, R3=BitCnt, R4=ByteCnt, R5=ATNCnt, R6=I2CCxt
985 ;*****
986 ; Handle DRDY
987 ;*****
02B1 EA 988 MASTER: MOV A,R2 ;Get data in A.
02B2 309D48 989 JNB I2CON.DRDY,MNDRDY ;Is it DRDY?
990 ; - context waiting to send bit
02B5 BE0418 991 CJNE R6,#TXBIT,NTxBit ;Context waiting to send bit?
02B8 F599 992 Tx1: MOV I2DAT,A ;Send bit
02BA 23 993 Tx2: RL A ;Rotate the byte.
02BB DB07 994 DJNZ R3,MN8Bit ;Was it 8th bit?
02BD 7E05 995 MOV R6,#TXREAD ;Set context waiting to read ACK
02BF 209D11 996 JB I2CON.DRDY,TxAck1 ;If DRDY, short cut.
02C2 21E5 997 AJMP WaitATN
998
999 ; prepare next bit 1-7
02C4 FA 1000 MN8Bit: MOV R2,A ;Put the data back.
02C5 209DF0 1001 JB I2CON.DRDY,Tx1 ;If DRDY, short cut.
02C8 209DED 1002 JB I2CON.DRDY,Tx1
02CB 209DEA 1003 JB I2CON.DRDY,Tx1 ;One more try.
02CE 21E5 1004 AJMP WaitATN
1005
1006 ; - context waiting to read ACK
02D0 BE050D 1007 NTxBit: CJNE R6,#TXREAD,NTxAck1 ;Context waiting to read ACK?
02D3 7598A0 1008 TxAck1: MOV I2CON,#CDR+CXA ;Switch to receive mode.
02D6 7E06 1009 MOV R6,#TXACK ;Set context waiting for ACK.
LOC OBJ LINE SOURCE

```


ACCESS.bus mouse application code for the
microcontroller

AN445

```

02D8 209D08      1010      JB      I2CON.DRDY,NTxAck2      ;If DRDY, short cut.
02DB 23          1011      RL      A                      ;Align data in case ARL.
02DC 1B          1012      DEC     R3                      ;
02DD FA          1013      MOV     R2,A                    ;
02DE 21E5        1014      AJMP    WaitATN                 ;
                                1015
                                1016      ; - context waiting for ACK
02E0 BE0629      1017      NTxAck1: CJNE R6,#TXACK,BeMast      ;Context waiting for ACK?
02E3 E598        1018      NTxAck2: MOV A,I2CON              ;Read from I2CON
02E5 5480        1019      ANL     A,#80h                  ;Only need 7th bit
02E7 6005        1020      JZ      AckOK                    ;
02E9             1021      BadAck: ;Stop if negative ACK.
02E9 D5097D      1022      DJNZ    NACnt,DoStp100K          ;Keep pending msg till retry expires.
02EC 6166        1023      AJMP    DoStp                     ;
                                1024      ;Ack Okay, prepare to send next byte.
02EE 0C          1025      AckOK: INC R4                      ;Point to byte we want to send.
                                1026      ;ACALL Sample
02EF 81B1        1027      AJMP    DOTXB                      ;
                                1028      ;return should AJMP DNTXB
02F1 EA          1029      DNTXB: MOV A,R2                    ;Next byte in A.
02F2 620D        1030      XRL     Check,A,A                ;XOR with message check.
02F4 7B08        1031      MOV     R3,#8                    ;BitCnt=8
02F6 7E04        1032      MOV     R6,#TXBIT                ;Set context to send bit.
02F8 209DBD      1033      JB      I2CON.DRDY,Tx1A          ;If DRDY, short cut.
02FB 21E5        1034      AJMP    WaitATN                 ;
                                1035
                                1036      $EJ
                                1037      ;*****
                                1038      A: ; It wasn't DRDY. Could be ARL, START, or STOP.
                                1039      ; ; If ARL, align data with tx bit that lost arbitration
                                1040      ; ; go to handle as SLAVE (MASTER was cleared).
                                1041      ; ; If it was START, we just became master.
                                1042      ; ; If STOP, not sure how this occurred,
                                1043      ; ; should have seen ARL, become idle SLAVE.
                                1044      ;*****
02FD 309C07      1045      MNDRDY: JNB I2CON.ARL,MNARL        ;ARL?
0300 03          1046      RR      A                      ;Rotate data back.
0301 0B          1047      INC     R3                      ;Increment bit count.
0302 03          1048      RR      A                      ;
0303 0B          1049      INC     R3                      ;
0304 FA          1050      MOV     R2,A                    ;
0305 4154        1051      AJMP    ARL0                     ;
0307 209B02      1052      MNARL: JB I2CON.STR,BeMast        ;
030A 4185        1053      AJMP    IdleS                    ;
                                1054
                                1055      ; - context START or not waiting for bit or ACK (from above).
                                1056      ; Must have just become master. Start new message.
                                1057      ; If message is pending in send type, do it.
                                1058      ; Otherwise, check to send Position Report if needed.
030C A90B        1059      BEMAST: MOV R1,SndType            ;Get message type.
030E B90043      1060      CJNE R1,#I_NoMsg,SndMsg          ;No message pending?
0311 302007      1061      JNB     MsgCheck,BeMast1         ;Send error?
0314 750BE4      1062      MOV     SndType,#I_Error          ;
0317 C220        1063      CLR     MsgCheck                ;
0319 6154        1064      AJMP    SndMsg                  ;
031B 201902      1065      BeMast1: JB SendRpt,PosMsg         ;Send position report?
031E 6166        1066      AJMP    DoStp                    ; then stop.
                                1067
                                1068      ; Send Position report
LOC  OBJ        LINE      SOURCE

```

ACCESS.bus mouse application code for the
microcontroller

AN445

```

0320 201C0C      1069      PosMsg: JB      TxSelfRst,PosMsg1      ;If first user data,
0323 D21C      1070      SETB      TxSelfRst      A      ;
0325 750BF0      1071      MOV      SndType,#I_Reset      ;Send a I_Reset to my address
0328 E508      1072      MOV      A,MyAddr      A,SX      ;
032A 750901      1073      MOV      NACnt,#1      ;No retry if not acknowledged.
032D 8027      1074      SJMP      SndMsg1      ;
032F 750B03      1075      PosMsg1:MOV      SndType,#LD_Position      ;Send position report.
0332 C219      1076      CLR      SendRpt      ;Clear pending report state
0334 E4      1077      CLR      A      ;Copy X-Y counts to xmt buffer.
0335 F512      1078      MOV      XBUF2,A      ;Hi byte=0.
0337 F514      1079      MOV      YBUF2,A      ;
0339 E516      1080      MOV      A,XCOUNT      ;Copy X.
033B F513      1081      MOV      XBUF1,A      ;
033D 30E703      1082      JNB      ACC.7,Copy2      ;If negative,
0340 7512FF      1083      MOV      XBUF2,#0FFh      ; extend sign.
0343 E517      1084      Copy2: MOV      A,YCOUNT      ;Copy Y.
0345 F515      1085      MOV      YBUF1,A      ;
0347 30E703      1086      JNB      ACC.7,Copy3      ;If negative,
034A 7514FF      1087      MOV      YBUF2,#0FFh      ; extend sign.
034D E4      1088      Copy3: CLR      A      ;
034E F516      1089      MOV      XCOUNT,A      ;Reset X-Y counters.
0350 F517      1090      MOV      YCOUNT,A      ;
0352 C21A      1091      CLR      Movement      ;Clear movement flag.
0354 7450      1092      SndMsg: MOV      A,#Adr_Host      ;
0356 F599      1093      SndMsg1:MOV      I2DAT,A      ;Send first bit by hand
0358 75981C      1094      MOV      I2CON,#CARL+CSTR+CSTP      ;Clear start, release SCL
0359      1095      ;Set context for rest of address
035B F50D      1096      MOV      Check,A      ;Init checksum
035D FA      1097      MOV      R2,A      ;I2Dat=A
035E 7B08      1098      MOV      R3,#8      ;BitCnt=8
0360 7C00      1099      MOV      R4,#0H      ;ByteCnt=0
0362 7E04      1100      MOV      R6,#TXBIT      ;Set context waiting to send bit
0364 41BA      1101      AJMP      Tx2      ;
0365      1102      $EJ      ;
0366      1103      ; Completed sending message, do STOP
0366 750B00      1104      DoStp: MOV      SndType,#I_NoMsg      ;Indicate cmd no longer pending.
0369 717B      1105      DoStp1: ACALL      SndStop      ;Send STOP.
036B E50B      1106      MOV      A,SndType      ;Is there a pending message?
036D 7006      1107      JNZ      DoStp3      ;
036F 201903      1108      JB      SendRpt,DoStp3      ;Is there a Position Report?
0372 302004      1109      JNB      MsgCheck,DoStp4      ;Is there an error message?
0375 11AF      1110      DoStp3: ACALL      SDelay      ;Yes, delay to give others
0376      1111      ; a chance to become master without contention.
0377 D2DE      1112      SETB      I2CFG.MASTRQB      ;Request to be master again.
0379 4199      1113      DoStp4: AJMP      RxStp0      ;Borrow code from receiver.
037A      1114      ; Send I2C STOP signal
037B C2DE      1115      SndStop:CLR      I2CFG.MASTRQB      ;Release Master request
037D 759821      1116      MOV      I2CON,#CDR+XSTP      ;Set to send stop
0380 309EFD      1117      JNB      I2CON.ATN,$      ;Wait for ATN
0383 759820      1118      MOV      I2CON,#CDR      ;Clear useless DRDY (rising SCL)
0386 309EFD      1119      JNB      I2CON.ATN,$      ;Wait for stop sent
0389 759894      1120      MOV      I2CON,#CARL+CSTP+CXA      ;Clear I2C bus
038C 7E01      1121      MOV      R6,#RXIDLE      ;Set context idle receiver.
038E 22      1122      RET      ;
038F      1123      RET      ;
0390      1124      $EJ      ;
0391      1125      ;*****
0392      1126      ;*****
0393      1127      ; DO_RX_BYTE
0394      1128      ;*****

```

LOC OBJ

LINE

SOURCE

ACCESS.bus mouse application code for the microcontroller

AN445

```

1128      ; Received a complete byte, already acknowledged.
1129      ; Examine context to decide what to do with it.
1130      ;
1131      ; Enter: R2 (I2CDat) is byte received.
1132      ; R4 is offset to byte just received.
1133      ; Exit: Command parameters saved as needed.
1134      ; Checksum verified. Valid commands executed.
1135      ; Return by AJMP DNRXB
1136      ;
1137      ; R2=I2CDat, R3=BitCnt, R4=ByteCnt, R5=ATNCnt, R6=I2CCxt
1138      ;*****
038F BC0002 1139      DORXB: CJNE    R4,#0,DoRx1      ;Is it Address? (ByteCnt=0)?
0392 4137    1140      AJMP    DNRXB
0394 BC0102 1141      DoRx1: CJNE    R4,#1,DoRx2      ;Is it source Addr? (ByteCnt=1)
0397 4137    1142      AJMP    DNRXB
1143      ;
1144      DoRx2: MOV     A,R2              ;Get byte.
0399 EA      1145      CJNE    R4,#2,DoRx3      ;Is it P+len?
039A BC020C 1146      RLC     A                  ;Rotate Prot bit into C.
039D 33      1147      MOV     Prot,C          ;Save it.
039E 9218    1148      MOV     A,R2              ;Get "len".
03A0 EA      1149      ANL     A,#07Fh          ;
03A1 547F    1150      ADD     A,#3              ;Add overhead.
03A3 2403    1151      MOV     MsgLen,A          ;Save message length.
03A5 F50C    1152      AJMP    DNRXB
03A7 4137    1153      ;
1154      DoRx3: CJNE    R4,#3,DoRx4      ;Is it Command byte?
03A9 BC0304 1155      MOV     RcvType,A          ;Save it
03AC F50A    1156      AJMP    DNRXB
03AE 4137    1157      SEJ     A
1158      ;
1159      ; Test for end of command
1160      ; If command has no data, byte offset 4 will be the checksum.
03B0 E50C    1161      DoRx4: MOV     A,MsgLen          ;Get message length.
03B2 B5040A 1162      CJNE    A,ByteCnt,ToMny        ;End of command?
1163      ; sets carry if MsgLen<ByteCnt
03B5 D21F    1164      SETB    AA                  ;Yes, do not Acknowledge more bytes.
03B7 E50D    1165      MOV     A,Check            ;Check in A
03B9 6002    1166      JZ     CheckOk          ;Bad check?
03BB 811F    1167      AJMP    RxErr
1168      ; Message check is Ok, dispatch valid commands
03BD 8125    1169      CheckOk: AJMP    DORXCMD      ;Return AJMP DNRXB
1170      ;
1171      ; Test for ByteCnt beyond message length
03BF 5002    1172      ToMny: JNC     DoDat          ;Too many bytes?
1173      ; Yes, just exit, negative acknowledge already sent.
03C1 4137    1174      DoRx9: AJMP    DNRXB
1175      ;
1176      ;
1177      ; Receive message data bytes
1178      ; ByteCnt from 4 to (MsgLen-1)
1179      ;
1180      ; Branch on RcvType to decide what to do with each byte.
1181      ; Notice Reset, Identify, and Poll have no data.
1182      ; Protocol: Assign New Address, Capabilities request
1183      ;
03C3 EA      1184      DoDat: MOV     A,R2              ;Get data again since DoRx4 wiped it.
03C4 AF0A    1185      MOV     R7,RcvType            ;Put RcvType in R7 so we can CJNE
03C6 3018F8 1186      JNB     Prot,DoRx9          ;Ignore device data stream.
LOC OBJ     LINE     SOURCE

```

ACCESS.bus mouse application code for the microcontroller

AN445

	1187	; Process C/S command bytes.	
	1188		
	1189	\$EJ	
	1190	;*****	
	1191	; DORXB: Assign R1 to point to byte 0	
	1192	; Compare incoming bytes with ID string (ByteCnt=4-29).	
	1193	; If not equal, set not equal bit and ignore.	
	1194	; Compare ByteCnt=30-31 with random number.	
	1195	; If not equal, ignore (become idle receiver).	
	1196	; Save ByteCnt=32, the address as parameter (R0).	
	1197	;*****	
	1198		
03C9 BFF225	1199	Do5: CJNE R7,#1,AsnAdr,Do6 ;Assign command?	
03CC BC040B	1200	CJNE R4,#4,Asn2 ;Start of ID string (ByteCnt=4)?	
03CF 7900	1201	MOV R1,#0 ;Initialize R1 is index	
	1202		
	1203	; BytCnt <= 29	
03D1 E9	1204	Asn4: MOV A,R1 ;Get offset	
03D2 B18C	1205	ACALL GET_ID	
03D4 09	1206	Asn5: INC R1 ;Increment for next	
03D5 B50215	1207	CJNE A,I2CDAT,AsnIg ;Compare to received byte	
03D8 4137	1208	AJMP DNRXB ;Ok so far	
	1209		
03DA BC1E02	1210	Asn2: CJNE R4,#30,Asn3 ;ByteCnt=30?	
03DD 790E	1211	MOV R1,#RandH ;Yes, set to read random#	
03DF 40F0	1212	Asn3: JC ASN4 ;Jump if less than 30	
	1213		
	1214	; ByteCnt >= 30	
03E1 BC2004	1215	CJNE R4,#32,Asn6 ;ByteCnt=32?	
03E4 A802	1216	MOV R0,I2CDAT ;Save new address in R0.	
03E6 4137	1217	AJMP DNRXB	
03E8 5003	1218	Asn6: JNC AsnIg ;Jump if ByteCnt>32.	
03EA E7	1219	MOV A,R1 ;Get byte of random #	
03EB 80E7	1220	SJMP Asn5 ;Steal code from above.	
	1221		
03ED D21E	1222	AsnIg: SETB NotMyID ;It's not for me	
03EF 4137	1223	AJMP DNRXB	
	1224		
	1225	; Application Capabilities Request	
03F1 BFF321	1226	Do6: CJNE R7,#1,CapReq,Do7 ;CapRequest?	
03F4 BC0403	1227	CJNE R4,#4,Cpr1 ;ByteCnt=4?	
03F7 EA	1228	MOV A,R2 ;Check Cap h pointer=0?	
03F8 7014	1229	JNZ Cpr4 ;No, reset length and offset to zero.	
03FA BC0516	1230	Cpr1: CJNE R4,#5,Cpr9 ;ByteCnt=5?	
	1231	;Yes, check for valid offset.	
03FD E518	1232	MOV A,CapOffset ;Compute next offset.	
03FF 2519	1233	ADD A,CapLen	
0401 B50202	1234	CJNE A,I2CDAT,Cpr2 ;Send next?	
0404 8111	1235	AJMP Cpr8 ; Yes, jump to set offset.	
0406 EA	1236	Cpr2: MOV A,R2 ;A=received offset.	
0407 6008	1237	JZ Cpr8 ;Send first?	
0409 B51802	1238	CJNE A,CapOffset,Cpr4 ;Send previous?	
040C 4137	1239	AJMP DNRXB ; Yes, use current offset.	
040E E4	1240	Cpr4: CLR A ;Reset Length and offset to zero.	
040F F519	1241	MOV A,CapLen	
0411 F518	1242	Cpr8: MOV CapOffset,A ;Load Cap offset.	
0413 4137	1243	Cpr9: AJMP DNRXB	
	1244		
	1245	; Set reporting Interval	
LOC OBJ	LINE	SOURCE	

ACCESS.bus mouse application code for the microcontroller

AN445

```

0415 BF8205 1246 Do7: CJNE R7,#LD_SetInterval,Do7a
0418 BC0402 1247 CJNE R4,#4,Do7a ;ByteCnt=4?
041B A802 1248 MOV R0,I2CDat ;Save parameter in R0.
041D 4137 1249 Do7a: AJMP DNRXB
1250
1251 ; Command error (I_Error)
041F D220 1252 RxErr: SETB MsgCheck ;Set to report error.
0421 D2DE 1253 SETB I2CFG.MASTRQB
0423 4137 1254 AJMP DNRXB
1255 $EJ
1256 ;*****
1257 ; DO_RX_CMD
1258 ; Valid command received, do it.
1259 ; Dispatch commands based on RcvType
1260 ; Parameter value is in R0.
1261 ;
1262 ; Commands Recognized: I_Reset, I_IdReq, I_AsgnAdr, I_CapReq,
1263 ; App_Test, App_Poll, App_SetInterval
1264 ;
1265 ; Return is: AJMP DNRXB. A is not preserved.
1266 ;*****
0425 AF0A 1267 DORXCMD:MOV R7,RcvType ;Put RcvType in R7 so we can CJNE.
0427 201802 1268 JB Prot,DoC4 ;Go for Control/Status commands.
042A 4137 1269 DoC4: AJMP DNRXB ;Ignore device data stream msgs.
1270 ;
1271 ;Bus protocol commands
1272 ; Reset
042C BFF002 1273 DoC4: CJNE R7,#I_Reset,DoC5
042F 01B4 1274 AJMP PwrUp ;Do power up reset.
1275 ; Identify
0431 BFF112 1276 DoC5: CJNE R7,#I_IdReq,DoC6
0434 750BE1 1277 MOV SndType,#I_IdReply ;Message type is identify
0437 201D08 1278 JB KeepID,RTBM ;Keep same device number?
043A D21D 1279 SETB KeepID
043C 858A0F 1280 MOV RandL,TL ;Random number <- T0
043F 858C0E 1281 MOV RandH,TH ;Random number <- T0
0442 D2DE 1282 RTBM: SETB I2CFG.MASTRQB ;Request to be master
0444 4137 1283 AJMP DNRXB
1284 ; Assign
0446 BFF20C 1285 DoC6: CJNE R7,#I_AsgnAdr,DoC7
0449 101E07 1286 JBC NotMyID,Nd01 ;Was it a complete match?
044C BC21D0 1287 CJNE R4,#33,RxErr ;Check len=30+3
044F C21C 1288 CLR TxSelfRst ;Anticipate first user data.
0451 8808 1289 MOV MyAddr,R0 ;Load new address
0453 4137 1290 Nd01: AJMP DNRXB
1291 ; Capabilities Request
0455 BFF30C 1292 DoC7: CJNE R7,#I_CapReq,DoC8
0458 BC0600 1293 CJNE R4,#6,$+3 ;Check len>=3+3
045B 40C2 1294 JC RxErr
045D 750BE3 1295 MOV SndType,#I_CapReply ;Message type is Cap Report
0460 D2DE 1296 SETB I2CFG.MASTRQB ;Request to be master
0462 4137 1297 AJMP DNRXB
1298 ; App Test
0464 BFB107 1299 DoC8: CJNE R7,#App_Test,DoC9
0467 750BA1 1300 MOV SndType,#App_TestReply ;Send a test report
046A D2DE 1301 SETB I2CFG.MASTRQB ;Request to be master.
046C 4137 1302 AJMP DNRXB
1303 ; App Poll
046E BFB00D 1304 DoC9: CJNE R7,#LD_Poll,DoC10
LOC OBJ LINE SOURCE

```

ACCESS.bus mouse application code for the microcontroller

AN445

```

0471 E508      1305      MOV     A,MyAddr      ;Check for default address.
0473 B46E02    1306      CJNE    A,#Adr_Default,DoC9a
0476 8004      1307      SJMP    DoC9b      ;Don't send Position reports
                                ; to default address.
0478 D219      1309      DoC9a: SETB    SendRpt      ;Flag to send Position report
047A D2DE      1310      SETB    I2CFG.MASTRQB      ;Request to be master.
047C 4137      1311      DoC9b: AJMP    DNRXB
                                ;
                                ; Set Locator report interval
047E BF822E    1314      DoC10: CJNE    R7,#LD_SetInterval,DoC11
0481 BC0500    1315      CJNE    R4,#5,$+3      ;Check: len>=2+3
0484 4099      1316      JC      RxErR      ;
0486 B80005    1317      CJNE    R0,#0,DoC10a      ;Check parameter range.
                                ; parameter=0, polling only.
0489 758800    1319      MOV     TCON,#0      ;Turn off Timer0.
048C 4137      1320      AJMP    DNRXB
048E B80800    1321      DoC10a: CJNE    R0,#8,$+3
0491 401A      1322      JC      DoC10c      ;Jump if R0<8.
0493 B81A00    1323      CJNE    R0,#26,$+3
0496 5015      1324      JNC     DoC10c      ;Jump if R0>=26.
                                ; 8 <= param <= 25, compute Timer0 reload value.
0498 74FE      1326      MOV     A,#0FFh      ;Start at FFFFh
049A F9        1327      MOV     R1,A
049B C3        1328      DoC10b: CLR     C      ;Loop to subtract
049C 949A      1329      SUBB    A,#MSECL      ; R0 milliseconds.
049E C9        1330      XCH     A,R1
049F 9402      1331      SUBB    A,#MSECH
04A1 C9        1332      XCH     A,R1
04A2 D8F7      1333      DJNZ    R0,DoC10b
04A4 F58B      1334      MOV     R1L,A      ;Set Timer0 reload.
04A6 898D      1335      MOV     R1L,A
04A8 758810    1336      MOV     TCON,#INIT_TCON      ;Turn on Timer0.
04AB 4137      1337      AJMP    DNRXB
04AD 811F      1338      DoC10c: AJMP    RxErR
                                ;
                                ; Unrecognized command (ignore it)
04AF 4137      1341      DoC11: AJMP    DNRXB
                                ;
                                ;
04B1 BC0104    1359      DOTXB: CJNE    R4,#1,DoTx1      ;ByteCnt=1?
04B4 AA08      1360      MOV     R2,MyAddr      ;Send source addr
04B6 41F1      1361      AJMP    DNTXB
                                ;
                                ; - Source address
04B8 41F1      1362
04B8 41F1      1363      ; - Message length
LOC OBJ      LINE      SOURCE

```


ACCESS.bus mouse application code for the
microcontroller

AN445

```

04B8 BC0258      1364      (DoTx1: CJNE R4,#2,DoTx2 ;ByteCnt=2?
04BB 7A82        1365      MOV R2,#082h ;Use 2 as default length,
                                ; P=1, Control/Status msg.
                                ;
04BD 750C05      1367      MOV R2,MsgLen,#5 ;Include overhead
                                ;
                                ; Compute length based on message type.
                                ;
                                ; If not Position Report, Identify, Output Error,
                                ; or Reset, the length is 2.
04C0 AF0B        1371      MOV R7,RndType
                                ;
                                ; Position report
04C2 BF0307      1373      CJNE R7,LD_Position,TxA ;Position report?
04C5 7A06        1374      MOV R2,#6 ;Data length is 6 (P=0; data stream).
04C7 750C09      1375      MOV R2,MsgLen,#9 ;6 plus 3 overhead.
04CA 41F1        1376      AJMP DNTXB
                                ;
                                ; Attention
04CC BFE00C      1378      TxA: CJNE R7,#I_Attn,TxI
04CF E51A        1379      MOV A,SelfTest ;Check for ROM error
04D1 B40105      1380      CJNE A,#ROM_ERROR,TxA9
04D4 7A83        1381      Len3: MOV R2,#083h ;Use Len=3 to include checksum.
04D6 750C06      1382      MOV R2,MsgLen,#6
04D9 41F1        1383      TxA9: AJMP DNTXB
                                ;
                                ; Identify Reply
04DB BFE107      1385      TxI: CJNE R7,#I_IdReply,TxC
04DE 7A9D        1386      MOV R2,#(80h+29) ;Length for Identify.
04E0 750C20      1387      MOV R2,MsgLen,#(29+3) ;Add overhead for MsgLen
04E3 41F1        1388      AJMP DNTXB
                                ;
                                ; Capabilities Report
04E5 BFE31C      1390      TxC: CJNE R7,#I_CapReply,TxE
04E8 7410        1391      MOV A,#CapFragLen ;Get default fragment length.
04EA F519        1392      MOV A,CapLen,A ;Save it.
04EC 2518        1393      ADD A,CapOffset ;Find end of fragment.
04EE C3          1394      CLR C
04EF 9475        1395      SUBB A,#(CAP_END-CAP_START) ;Is it beyond end of Cap String?
04F1 4006        1396      JC TxC3 ;No, use default length.
04F3 F4          1397      TxCl: CPL A ;Yes, shorten as needed.
04F4 04          1398      INC A
04F5 2519        1399      ADD A,CapLen
04F7 F519        1400      MOV A,CapLen
04F9 E519        1401      TxC3: MOV A,CapLen ;Get fragment length.
04FB 2483        1402      ADD A,#083h ;Compute data length.
04FD FA          1403      MOV R2,A ;Prepare to send it.
04FE 2483        1404      ADD A,#083h ;Add 3 overhead for MsgLen (clear C/S).
0500 F50C        1405      MOV R2,MsgLen,A
0502 41F1        1406      AJMP DNTXB
                                ;
                                ; Checksum or message framing error
0504 BFE402      1408      TxE: CJNE R7,#I_Error,TxR
0507 8003        1409      SJMP TxR1
                                ;
                                ; Reset
0509 BFF005      1411      TxR: CJNE R7,#I_Reset,TxU
050C 7A81        1412      TxR1: MOV R2,#081h ;Length for Reset (80+1).
050E 750C04      1413      MOV R2,MsgLen,#4 ;1 plus 3 overhead.
0511 41F1        1414      TxU: AJMP DNTXB
                                ;
                                ; - Command code
0513 BC0309      1417      DoTx2: CJNE R4,#3,DoTxLast ;ByteCnt=3?
0516 AA0B        1418      MOV R2,RndType ;Send command code
0518 BA0302      1419      CJNE R2,LD_Position,TCC1 ; unless it is position report.
051B AA10        1420      MOV R2,ReportBuf ;In that case send 1st byte of report.
051D 41F1        1421      TCC1: AJMP DNTXB
                                ;
                                ;
051F 41F1        1422      ;
LOC OBJ          LINE      SOURCE

```

	1423		; - Test for last byte of command (message check)	
051F E50C	1424	DoTxLast: MOV	A,MsgLen	
0521 B50404	1425	CJNE	A,ByteCnt,DoTxEnd ;Last byte of command?	
	1426		; sets Carry if A<ByteCnt	
0524 AA0D	1427	MOV	R2,Check ;Yes, send check.	
0526 41F1	1428	AJMP	DNTXB	
	1429			
	1430		; - Test for beyond last byte of command	
0528 5005	1431	DoTxEnd: JNC	DoTx3 ;Beyond last byte (check)?	
052A 750905	1432	MOV	NACnt,#5 ;Reset Negative Ack retry count.	
052D 6166	1433	AJMP	DoStp ;Send STOP.	
	1434			
	1435			
	1436		; Transmit message data bytes	
	1437		; ByteCnt from 3 to (length+2)	
	1438		; Dispatch based on command type	
	1439			
052F AF0B	1440	DoTx3: MOV	R7,SndType	
0531 BF0309	1441	CJNE	R7,#LD_Position,DoT3 ;Position report?	
0534 E504	1442	MOV	A,ByteCnt ;Yes, send next byte.	
0536 240D	1443	ADD	A,#ReportBuf-3	
0538 F9	1444	MOV	R1,A;eqip	
0539 8702	1445	MOV	I2CDat,@R1 ;R2=I2CDat=@R1	
053B 41F1	1446	AJMP	DNTXB	
	1447			
	1448		;Attention report	
053D BFE009	1449	DoT3: CJNE	R7,#I_Attn,DoT4	
0540 AA1A	1450	MOV	R2,SelfTest ;Send Power-up selftest and attention	
0542 BC0502	1451	CJNE	R4,#5,AT4 ;ByteCnt=5?	
0545 AA1B	1452	MOV	R2,RomSum ;Yes, send checksum byte.	
0547 41F1	1453	AT4: AJMP	DNTXB	
	1454			
	1455		;Application test report	
0549 BFA104	1456	DoT4: CJNE	R7,#App_TestReply,DoT5	
054C AA1A	1457	MOV	R2,SelfTest ;Send Selftest result	
054E 41F1	1458	AJMP	DNTXB	
	1459			
	1460		*****	
	1461		; Identify report	
	1462		; Send ID string for ByteCnt 4-29 (26 bytes, last two are FFh).	
	1463		; Send random number for ByteCnt 30 and 31.	
	1464		*****	
0550 BFE11D	1465	DoT5: CJNE	R7,#I_IdReply,DoT6	
0553 BC0409	1466	CJNE	R4,#4,IDR2 ;First byte (ByteCnt=4)?	
	1467		;yes, set up to send ID string	
0556 7900	1468	MOV	R1,#0 ;R1 is index	
	1469			
0558 E9	1470	IDR4: MOV	A,R1 ;Get offset	
0559 B18C	1471	ACALL	GET_ID	
055B FA	1472	IDR5: MOV	R2,A ;Prepare to send it	
055C 09	1473	INC	R1 ;Increment for next	
055D 41F1	1474	AJMP	DNTXB	
	1475			
055F BC1E02	1476	IDR2: CJNE	R4,#30,IDR3 ;ByteCnt=30?	
0562 790E	1477	MOV	R1,#RandH ;Set to send random #	
0564 40F2	1478	IDR3: JC	IDR4 ;Jump if less than 30	
	1479		; BytCnt >= 30	
0566 BC2000	1480	CJNE	R4,#32,\$+3	
0569 5003	1481	JNC	IDR7 ;Jump if >= 32	
LOC OBJ	LINE	SOURCE		

ACCESS.bus mouse application code for the
microcontroller

AN445

```

056B E7          1482      MOV     A,@R1      ;Get byte of random #1
056C 80ED        1483      SJMP    IDR5
056E 6166        1484      IDR7:  AJMP    DoStop ;Error! Beyond last byte,
                                ; STOP now.
                                ;*****
                                ; Capability report
                                ; Send next byte of capability string.
                                ; Uses R1 as index.
                                ;*****
0570 BFE317      1492      DoT6:  CJNE    R7,#I_CapReply,DoT7
0573 BC0404      1493      CJNE    R4,#4,Cap1 ;First byte (ByteCnt=4)?
0576 7A00        1494      MOV     R2,#0      ;Send High byte of Offset (always 0)
0578 41F1        1495      AJMP    DNTXB
                                ;*****
057A BC0506      1497      Cap1:  CJNE    R4,#5,CAP2      ;Second byte (Offsetlo)?
057D AA18        1498      MOV     R2,CapOffset ;Send Low byte of Offset.
057F A918        1499      MOV     R1,CapOffset ;Initialize R1 to use as index.
0581 41F1        1500      AJMP    DNTXB
                                ;*****
0583 E9          1502      Cap2:  MOV     A,R1 ;Get Capabilities Character.
0584 B1A9        1503      ACALL  GET_CAP
0586 FA          1504      MOV     R2,A      ;Prepare to send it.
0587 09          1505      INC     R1      ;Increment for next Character.
0588 41F1        1506      Cap3:  AJMP    DNTXB
                                ;*****
                                ; Unknown: How can we not know what we're sending?
058A 41F1        1509      DoT7:  AJMP    DNTXB
                                ;*****
                                ; $EJ
                                ;*****
                                ; GET_ID
                                ; Get byte of ID string
                                ; Enter: offset of desired byte in A.
                                ; Exit: A is the desired byte.
                                ;*****
058C 04          1519      GET_ID: INC     A ;Skip RET
058D 83          1520      MOVC    A,@A+PC ;Get the byte
058E 22          1521      RET
                                ;ID string is defined here, length is 25
058F 41          1523      DB      'A' ;Protocol revision
0590 56312E31    1524      DB      'V1.1 ' ;Module revision
0594 202020      1525      DB      'DEC ' ;Vendor name
0597 44454320    1526      DB      'VSXXX-BB' ;Module name
059B 20202020
059F 56535858
05A3 582D4242
05A7 FF          1527      DB      0FFh ;1st byte of device #
05A8 FF          1528      DB      0FFh ;2nd byte of device #
                                ;*****
                                ; GET_CAP
                                ; Get byte of Capabilities string.
                                ; This implementation supports up to 254 bytes only!
                                ; Enter: offset of desired byte in A.
                                ; Exit: A is the desired byte.
                                ;*****
05A9 04          1537      GET_CAP: INC     A ;Skip RET
LOC OBJ          1537      SOURCE

```

ACCESS.bus mouse application code for the microcontroller

AN445

05AA 83	1538	MOVC A, @A+PC ;Get the byte	1483	0253 83
05AB 22	1539	RET	1484	0254 83
	1540	;Capabilities string is defined here.	1485	0255 83
05AC 28	1541	CAP_START: DB ' ('	1486	
05AD 2070726F	1542	DB ' prot(locator)'	1487	
05B1 74286C6F			1488	
05B5 6361746F			1489	
05B9 7229			1490	
05BB 20747970	1543	DB ' type(mouse)'	1491	
05BF 65286D6F			1492	
05C3 75736529			1493	
05C7 20627574	1544	DB ' buttons(1(L)2(R)3(M))'	1494	
05CB 746F6E73			1495	
05CF 2831284C			1496	
05D3 29322852			1497	
05D7 2933284D			1498	
05DB 2929			1499	
05DD 2064696D	1545	DB ' dim(2)'	1500	
05E1 283229			1501	
05E4 2072656C	1546	DB ' rel'	1502	
05E8 20726573	1547	DB ' res(200 inch)'	1503	
05EC 28323030			1504	
05F0 20696E63			1505	
05F4 6829			1506	
05F6 2072616E	1548	DB ' range(-127,127)'	1507	
05FA 6765282D			1508	
05FE 31323720			1509	
0602 31323729			1510	
0606 20643028	1549	DB ' d0(dname(X))'	1511	
060A 646E616D			1512	
060E 65285829			1513	
0612 29			1514	
0613 20643128	1550	DB ' d1(dname(Y))'	1515	
0617 646E616D			1516	
061B 65285929			1517	
061F 29			1518	
0620 29	1551	DB ')'	1519	
0621 00	1552	CAP_END: DB 0 ;Null terminator (not used).	1520	
	1553	; Capabilities length is 121 bytes	1521	
	1554		1522	
	1555	END	1523	
			1524	
			1525	
			1526	
			1527	
			1528	
			1529	
			1530	
			1531	
			1532	
			1533	
			1534	
			1535	
			1536	
			1537	
			1538	
			1539	
			1540	
			1541	
			1542	
			1543	
			1544	
			1545	
			1546	
			1547	
			1548	
			1549	
			1550	
			1551	
			1552	
			1553	
			1554	
			1555	

A software duplex UART for the 751/752 AN446

Author: Greg Goodhue

The following program contains routines that will allow an 8xC751 or 8xC752 to implement a software UART that can send and receive serial data simultaneously. Other published software UARTs only allow either transmit or receive to occur at any one time. The demo application shown in the code listing waits for data to be received, then echoes it and follows this with a hexadecimal interpretation of the data plus a space. For instance, if the program receives the character "\$", it echoes back the string "\$24". The reason for echoing these additional characters is to make it easy to force the receiver buffer to fill up in order to test the handshaking. If the program simply echoed what was received, it would likely never use more than the very first receiver buffer location since it can normally transmit just as fast as it can receive.

CHIP RESOURCES

The UART routines use about 400 bytes of code space and use the timer to provide a constant time interrupt to synchronize both transmit and receive operations. The hardware connections require four device pins to accomplish serial I/O with RTS/CTS handshaking. Only two pins would be needed if handshaking is not required. Three of the four pin functions may be assigned to any port pin. The serial input pin must be assigned as one of the external interrupt pins. Another two pins are used in the demo application to input a selection of one of four baud rates (1200, 2400, 4800, or 9600).

LIMITATIONS

To obtain duplex operation, a fairly large portion of the chip's time is used. The routines were tested up to 9600 baud running on a 16 MHz 87C751. When serial input and output were both occurring at the same time, the routines could not support continuous operation with no pauses between characters. At 4800 baud, full speed tight reception and transmission worked flawlessly. In other words, 4800 baud should work with all applications, while 9600 baud may not work with all applications.

THEORY OF OPERATION

There are three possible sequences of events when serial transmit and receive may both be operating at once: transmit and receive begin simultaneously; transmit is requested while the receiver is busy; and receive starts while the transmitter is busy. The first 2 cases could be handled fairly simply with only one interrupt for each bit time. In the first case, everything is already in synch and only one

timer and one interrupt per bit is needed to do both operations. In the second case (transmit is requested while the receiver is busy) the program could just wait for the next bit time to start transmitting. Unfortunately, the third case presents a problem. If the program is already transmitting, it cannot always wait for the next bit time to start sampling the serial data if the application is not to lose bits. Also, the timer cannot be adjusted to the incoming data since this would distort the duration of one of the transmitted bits.

The method used here to deal with this problem is to always divide all bit times into 4 sub-bit times. When transmission and/or reception is in progress, the timer runs at 4X the bit rate for the selected baud rate. The variables TxTime and RxTime are used to count sub-bit times for the transmitter and the receiver, respectively. Both are initialized to a negative value and count up to simplify testing for an active sub-bit time. The maximum baud rate that can be supported is essentially determined by the maximum amount of time that it might take the microcontroller to do all of the operations associated with transmitting one bit and receiving one bit. This must be done within the time between timer interrupts.

When both transmit and receive operations are scheduled for the same timer interrupt, priority is given to the transmitter routine. The reason for this is that a great deal of jitter can be tolerated in the timing of the received bit sampling, but the transmitted data must "look" good to the outside world.

The actual bit times for transmit and receive are counted by the variables TxCnt and RxCnt, respectively. When an active sub-bit time slice occurs, these variables tell the transmit and receive routines what to do in the current time slice. The value 11 hex indicates a start bit, 10 hex indicates a stop bit, and the values 8 through F hex indicate a data bit. The values were chosen to allow quick determination of the appropriate action by the code.

The routines provide for a small amount of data buffering for both the transmitter and the receiver. As implemented here, the transmitter buffer is only one byte deep, allowing one data byte to be held while another is being transmitted. The receiver buffer is larger, allowing three bytes to be held while a fourth is being received. If the receiver buffer fills up (indicated by the flag RxFull), the application code must retrieve one byte before a fourth one finishes, or data will be lost. If this happens, a flag will be set (OverrunErr) to indicate that the receiver buffer has been overrun. There is no similar flag for the transmitter, since the transmit

request routine waits for the transmitter buffer to be available (indicated by the TxFull flag) before taking action. It is up to the application code to check this flag in advance if it does not want to stall execution while waiting to transmit data.

As each routine finishes a whole data byte by completing the send or receive of a stop bit, it checks to see if there is something still happening to warrant having the time slice interrupt running. In the case of a received stop, the transmit activity flag (TxOn) is examined. If it is not set, the timer is turned off. The timer will be turned back on if an interrupt from a serial start bit is received or the main code requests data to be transmitted. In the case of a transmitted stop, both the receiver activity flag (RxOn) and the transmit buffer flag (TxFull) are examined. If the receiver is active or there is more data to transmit, the timer is left running.

All of the status flags are in the "Flags" register. Other status flags found there are: RxAvail, which indicates that the receiver buffer contains unprocessed data; and FramingErr which is set when the receiver routines find an improper start or stop bit, usually caused by mismatched baud rates.

Flow control handshaking is provided by the RTS/CTS scheme. The transmit routine looks at the incoming CTS line before beginning each start bit transmission, and simply exits, waiting for the next time slice, if CTS is not asserted. The receive routine checks the buffer status whenever a start bit interrupt occurs and de-asserts the outgoing RTS line if the buffer already contains two bytes (i.e., it will be full when the current byte finishes). If the device at the other end of the communication line follows the same rules (which may very well NOT be the case) the program should be able to communicate without buffer overflows in either direction.

Baud rates in both the send and receive routines are determined by two things: the timer interrupt rate; and the number of time slices per bit. The method of calculating the timer value for various baud rates is discussed in the code listing at the BaudRate routine. This discussion has centered on there being four time slices per bit, but if the user wants, either the transmitter or the receiver can be set to run at a baud rate that is a multiple of the other by adjusting the value of the constant TxBitLen or RxBitLen. The baud rate would be calculated as indicated for the faster channel, and TxBitLen or RxBitLen would be changed for the slower channel. For example, the transmitter can be set to run at half of the receiver baud rate by setting TxBitLen to $-8 + 1$.

The routines shown also make provision for changing the baud rate "on the fly", although the application code given does not implement this feature. If the application code changes the baud rate for some reason, the change will be effected when the next data transmission or reception begins, if both the transmitter and receiver were already idle. This prevents the timer value from being changed in the middle of a data byte.

THE CODE

There are a number routines in the code that the user should be aware of:

- **Intr0**—Called (by interrupt) when a serial start bit is received.
- **Timer0**—Called (by interrupt) for every sub-bit time slice.
- **RS232TX**—Called by Timer0 when the transmitter has business to conduct in the current time slice.
- **RS232RX**—Called by Timer0 when the receiver has business to conduct in the current time slice.
- **BaudRate**—Sets the baud rate variables **BaudHigh** and **BaudLow** based on the accumulator value.
- **TxSend**—Called by the application code request that a data byte be transmitted.

The data to be transmitted is in the accumulator.

- **GetRx**—Called by the application code to request return of a received data byte from the buffer. Data is returned in the accumulator. This routine should not be called unless the receiver buffer has data available.

- **Reset**—Start of the initialization code to set up the UART.

- **MainLoop**—Start of the mainline code of the demo application.

A software duplex UART for the 751/752 AN446

```

;*****
;      Duplex UART Routines for the 8xC751 and 8xC752 Microcontrollers
;*****

; This is a demo program showing a way to perform simultaneous RS-232
; transmit and receive using only one hardware timer.

; The transmit and receive routines divide each bit time into 4 slices to
; allow synchronizing to incoming data that may be out of synch with outgoing
; data.

; The main program loop in this demo processes received data and sends it
; back to the transmitter in hexadecimal format. This insures that we can
; always fill up the receiver buffer (since the returned data is longer than
; the received data) for testing purposes. Example: if the letter "A" is
; received, we will echo "A41.".

;*****

$Title(Duplex UART Routines for the 751/752)
$Date(8/20/92)
$MOD751

;*****
;      Definitions
;*****

; Miscellaneous

TxBitLen EQU -4 + 1 ; Timer slices per serial bit transmit.
RxBitLen EQU -4 + 1 ; Timer slices per serial bit receive.
RxHalfBit EQU (RxBitLen / 4) + 1 ; Timer slices for a partial bit time.
; Used to adjust the input sampling
; time point.

; Note: TxBitLen and RxBitLen are kept separate in order to facilitate the
; possibility of having different transmit and receive baud rates. The timer
; would be set up to give four slices for the fastest baud rate, and the
; BitLen for the slower channel would be set longer for the slower baud rate.
; BitLen = -4 + 1 gives four timer interrupts per bit. BitLen = -8 + 1 would
; give 8 slices, BitLen = -16 + 1 would give 16 slices, etc.

TxPin BIT P1.0 ; RS-232 transmit pin (output).
RxPin BIT P1.5 ; RS-232 receive pin (input).
RTS BIT P1.3 ; RS-232 request to send pin (output).
CTS BIT P1.6 ; RS-232 clear to send pin (input).
; Note: P1.1 and P1.2 are used to input the baud rate selection.

; RAM Locations

Flags DATA 20h ; Miscellaneous bit flags (see below).
TxOn BIT Flags.0 ; Indicates transmitter is on (busy).
RxOn BIT Flags.1 ; Indicates receiver is on (busy).
TxFull BIT Flags.2 ; Transmit buffer (1 byte only) is full.
RxFull BIT Flags.3 ; Receiver buffer is full.
RxAvail BIT Flags.4 ; RX buffer is not empty.
OverrunErr BIT Flags.6 ; Overrun error flag.

```

A software duplex UART for the 751/752 AN446

```

FramingErr  BIT    Flags.7      ; Framing error flag.

BaudHigh    DATA  21h          ; High byte timer value for baud rate.
BaudLow     DATA  22h          ; Low byte timer value for baud rate.

TxCnt       DATA  23h          ; RS-232 byte transmit bit counter.
TxTime      DATA  24h          ; RS-232 transmit time slice count.
TxShift     DATA  25h          ; Transmitter shift register.
TxDat       DATA  26h          ; Transmitter holding register.

RxCnt       DATA  27h          ; RS-232 byte receive bit counter.
RxTime      DATA  28h          ; RS-232 receive time slice count.
RxShift     DATA  29h          ; Receiver shift register.
RxDatCnt    DATA  2Ah          ; Received byte count.
RxBuf       DATA  2Bh          ; Receive buffer (3 bytes long).

Temp        DATA  2Ph          ; Temporary holding register.

;*****
;
;***** Interrupt Vectors *****
;*****
ORG 00h          ; Reset vector.
AJMP RESET

ORG 03h          ; External interrupt 0
AJMP Intr0       ; (received RS-232 start bit).

ORG 0Bh          ; Timer 0 overflow interrupt.
AJMP Timer0      ; (4X the RS-232 bit rate).

ORG 13h          ; External interrupt 1.
RETI             ; (not used).

ORG 1Bh          ; Timer 1 interrupt.
RETI             ; (not used).

ORG 23h          ; I2C interrupt.
RETI             ; (not used).

;*****
;***** Interrupt Handlers *****
;*****
; External Interrupt Int0.
; RS-232 start bit transition.
Intr0:  PUSH  ACC      ; Save accumulator.
        PUSH  PSW      ; and status.
        CLR   IE.0     ; Disable more RX interrupts.

        SETB  RxOn      ; Set receive active flag.
        MOV  RxCnt,#11h ; Set bit counter to expect a start.
        MOV  RxTime,#RxHalfBit ; First sample is at a partial bit time.
        JB   TxOn,IOTimerOn ; If TX active then timer is on.

        MOV  RTH,BaudHigh ; Set up timer for selected baud rate.
        MOV  RTL,BaudLow  ; Receiver buffer is full.
        MOV  TH,BaudHigh  ; RX buffer is not empty.
        MOV  TL,BaudLow   ; Overrun error flag.

```

A software duplex UART for the 751/752 AN446

```

SETB TR ; Start timer 0.
; Timer 0 Interrupt
; This is used to generate time slices for both serial transmit and receive
; functions.

Timer0: PUSH ACC ; Save accumulator,
        PUSH PSW ; and status.
        JNB TxTime.7,RS232TX ; Is this an active time slice
        ; for an RS-232 transmit?
        JNB TxOn,CheckRx ; If transmit is active,
        INC TxTime ; increment the time slice count.
CheckRx: JNB RxTime.7,RS232RX ; Is this an active time slice
        ; for an RS-232 receive?
        JNB RxOn,T0Ex ; If receive is active, increment
        INC RxTime ; the time slice count.

T0Ex: POP PSW ; Restore status,
      POP ACC ; and accumulator.
      MOV P3,Flags ; For demo purposes, output status
      ; on an extra port.

      RETI

;*****
; RS-232 Transmit Routine
;*****

RS232TX: JNB TxCnt.4,TxData ; Go if data bit.
        JNB TxCnt.0,TxStop ; Go if stop bit.
; Send start bit and do buffer housekeeping.
TxStart: JB CTS,TxEx1 ; Is CTS asserted (low) so can we send?
        ; If not, try again after 1 bit time.
        CLR TxPin ; Set start bit.
        MOV TxShift,TxDat ; Get byte to transmit from buffer.
        CLR TxFull ;
        MOV TxCnt,#08h ; Init bit count for 8 bits of data.
        ; (note: counts UP).
TxEx1: MOV TxTime,#TxBitLen ; Reset time slice count.
      SJMP CheckRx ; Restore state and exit.

; Send Next Data Bit.
TxData: MOV A,TxShift ; Get un-transmitted bits.
      RRC A ; Shift next TX bit to carry.
      MOV TxPin,C ; Move carry out to the TXD pin.
      MOV TxShift,A ; Save bits still to be TX'd.

```

A software duplex UART for the 751/752 AN446

```

        INC     TxCnt           ; Increment TX bit counter
        MOV     TxTime,#TxBitLen ; Reset time slice count.
        SJMP    CheckRx        ; Restore state and exit.

; Send Stop Bit and Check for More to Send.

TxStop:  SETB    TxBuf         ; Send stop bit.
        JB      TxFull,TxEx2    ; More data to transmit?
        CLR     TxOn           ; If not, turn off TX active flag, and
        CLR     RTS           ; make sure that whoever is on the
                                ; other end knows it's OK to send.

        JB      RxOn,TxEx2      ; If receive active, timer stays on,
        CLR     TR             ; otherwise turn off timer.

TxEx2:   MOV     TxCnt,#11h      ; Set TX bit counter for a start.
        MOV     TxTime,#TxBitLen-1 ; Reset time slice count, stop bit
                                ; > 1 bit time for synch.
        SJMP    CheckRx        ; Restore state and exit.

;*****
;                               RS-232 Receive Routine
;*****

RS232RX: MOV     C,RxPin        ; Get current serial bit value.
        JNB     RxCnt.4,RxData  ; Go if data bit.
        JNB     RxCnt.0,RxStop  ; Go if stop bit.

;Verify start bit.

RxStart: JC      RxErr         ; If bit=1, then not a valid start.
        MOV     RxCnt,#08h      ; Init counter to expect data.
        MOV     RxTime,#RxBitLen ; Reset time slice count.
        SJMP    TOEx          ; Restore state and exit.

; Get Next Data Bit.

RxData:  MOV     A,RxShift      ; Get partial received byte.
        RRC     A              ; Shift in new received bit.
        MOV     RxShift,A      ; Store partial result in buffer.
        INC     RxCnt          ; Increment received bit count.
        MOV     RxTime,#RxBitLen ; Reset time slice count.
        SJMP    TOEx          ; Restore state and exit.

; Store Data Byte, "push"ing it into the FIFO buffer.

RxStop:  CLR     EA            ; Don't interrupt the following.
        MOV     A,RxBuf        ; "PUSH" the receive buffer.
        XCH     A,RxBuf+1      ;
        XCH     A,RxBuf+2      ;
        MOV     RxBuf,RxShift  ; Add just completed data to buffer.
        INC     RxDatCnt       ; Increment the received byte count.
        SETB    EA            ; Re-enable interrupts.

        SETB    RxAvail        ; There is data in the RX buffer.
        PUSH    PSW            ; Save Carry (received bit) for later.
        MOV     A,RxDatCnt     ; Check receiver buffer status.

```

A software duplex UART for the 751/752 AN446

```

CJNE A,#4,RxChk1      ; Is RX buffer overrun?
SETB OverrunErr        ; Set status reg overrun error flag.
MOV RxDatCnt,#3        ; Re-set buffer counter to "full".

RxChk1: CJNE A,#3,RxChk2      ; Is RX buffer full?
SETB RxFull            ; Set buffer full status.

RxChk2: POP PSW          ; Retrieve last received bit in Carry.
JC RxEEx              ; If bit=0, then not a valid stop.
RxErr: SETB FramingErr    ; Remember bad start or stop status.

RxEEx: JB TxOn,RxTimerOn  ; If transmit active, timer stays on,
CLR TR                ; otherwise turn timer off.
RxTimerOn: CLR RxOn      ; Turn off receive active.
SETB RxTime.7         ; Set bit for no service to
                        ; RX Time Slice Branches.
                        ; Re-enable RS-232 receive interrupts.
                        ; Restore state and exit.
SETB IE.0
AJMP TOEx

;*****
; Subroutines
;*****

; BaudRate - Determine and set the baud rate from switches.
; Note: if the baud rate is altered, the actual change will only occur when
; a transmit or receive is begun while the timer was not already running
; (i.e.: not already busy transmitting or receiving).

BaudRate: MOV DPTR,#BaudTable ; Set pointer to baud rate table.
ANL A,#03h ; Limit displacement for lookup.
RL A ; Double the table index since these
      ; are 2 byte entries.
      ; Save the table index for second byte.
      ; Get first byte, and save as the high
      ; byte of the baud rate timer value.
      ; Get back the table index.
      ; Advance to next table entry.
      ; Get second byte, and save as the low
      ; byte of the baud rate timer value.
      ; Ret

; Entries in BaudTable are for a timer setting of 1/4 of a bit time at the given
; baud rate. The two values per entry are the high and low bytes of the value
; respectively.

; Values are calculated as follows:
;
; Osc Frequency
; 1/4 Bit cell time (in machine cycles) = -----
; Baud Rate * 48
;
; Example for 9600 baud with a 16MHz crystal:
; 16,000,000 / 9600 * 48 = 34.7222... machine cycles per quarter bit time.
; Rounded, this is 35. The hexadecimal value for 35 is 23.
; 10000 hex - 23 hex (truncated to 16 bits) = FFDD. Thus, the BaudTable entry
; for 9600 baud is FF, DD hex.

BaudTable: DB 0FEh,0EAh ; 1200 baud.
            DB 0FFh,75h ; 2400 baud.
            DB 0FFh,0BBh ; 4800 baud.
            DB 0FFh,0DDh ; 9600 baud.

```

A software duplex UART for the 751/752 AN446

```

; TxSend - Initiate RS-232 Transmitt.
TxSend:    JB    TxFull,$      ; Make sure TX buffer is free.
           SETB   TxFull      ; Reserve the buffer for our use.
           MOV    TxDat,A      ; Put character in buffer.
           JB     TxOn,TSTimerOn ; Exit if transmitter already running.

           SETB   TxOn        ; Transmit active flag set.
           MOV    TxCnt,#11h    ; Init bit counter to expect a start.
           MOV    TxTime,#TxBitLen ; Reset time slice count.
           JB     RxOn,TSTimerOn ; Exit if receiver already active.

           MOV    RTH,BaudHigh  ; Set up timer for selected baud rate.
           MOV    RTL,BaudLow   ; Turn off receive active.
           MOV    TH,BaudHigh   ; Set bit for no receive.
           MOV    TL,BaudLow    ; RX Time Slice Branches.
           SETB   TR            ; Start up the bit timer.
TSTimerOn: RET

; PrByte - Output a byte as ASCII hexadecimal format.
PrByte:    PUSH   ACC          ; Print ACC contents as ASCII hex.
           SWAP   A
           ACALL  HexAsc      ; Print upper nibble.
           ACALL  TxSend
           POP    ACC
           ACALL  HexAsc      ; Print lower nibble.
           ACALL  TxSend
           RET

; HexAsc - Convert a hexadecimal nibble to its ASCII character equivalent.
HexAsc:    ANL    A,#0Fh      ; Make sure we're working with only
                           ; one nibble.
           CJNE   A,#0Ah,HAL   ; Test value range.
           JC     HAVal09      ; Value is 0 to 9.
           ADD    A,#7         ; Value is A to F, needs pre-adjustment.
           HAVal09: ADD    A,#'0' ; Adjust value to ASCII hex.
           RET

; GetRx - Retrieve a byte from the receive buffer, and return it in A.
GetRx:     CLR    EA          ; Make sure this isn't interrupted.
           DEC    RxDatCnt     ; Decrement the buffer count.
           MOV    A,RxDatCnt   ; Get buffer count.
           JNZ    GRX1        ; Test for empty receive buffer.
           CLR    RxAvail      ; If empty, clear data available status.
           GRX1:  ADD    A,#RxBuf ; Create a pointer to end of buffer.
           MOV    Temp,R0      ; Save R0.
           MOV    R0,A         ; Put pointer where we can indirect.
           MOV    A,@R0        ; Get last buffer data.
           MOV    R0,Temp      ; Restore R0.
           CLR    RxFull      ; Buffer can't be full anymore.
           SETB   EA          ; Re-enable interrupts.
           RET

;*****
;                               Reset
;*****
Reset:     MOV    SP,#2Fh      ; Initialize stack start.

```

A software duplex UART for the 751/752

AN446

```
MOV   TCON,#0           ; Set timer off, INTO to level trigger.
MOV   P3,#0             ; Turn off all status outputs.
```

```
; For this demo, we only set up the baud rate once at reset:
```

```
MOV   A, P1             ; Read baudrate bits from P1.
RR    A                 ; The switches are on bits 2 and 1.
ACALL BaudRate          ; Set up the selected baud rate.
```

```
MOV   FLAGS,#0          ; Init all status flags.
MOV   RxDatCnt,#0        ; Clear buffer count.
MOV   IE,#93h           ; Turn on timer 0 interrupt and
                        ; external interrupt 0.
CLR   RTS               ; Assert RTS so we can receive.
```

```
; The main program loop processes received data and sends it back to the
; transmitter in hexadecimal format. This insures that we can always fill
; up the receiver buffer (since the returned data is longer than the
; received data) for testing purposes. Example: if the letter "A" is
; received, we will echo "A41 ".
```

```
MainLoop: JNB  RxAvail,$   ; Make sure an input byte is available.
          ACALL GetRx      ; Get data from the receiver buffer.
          ACALL TxSend     ; Echo original character.
          ACALL PrByte     ; Output the char in hexadecimal format,
          MOV   A,#20h     ; followed by a space.
          ACALL TxSend
          SJMP  MainLoop   ; Repeat.
```

```
END
```


Section 3

ACCESS.bus Application Notes & Articles

INDEX

AN445	ACCESS.bus mouse application code for the microcontroller	See Section 2
Serial bus looks to standardize connections for peripherals		3-3
ACCESS.bus, an Open Desktop Bus		3-5
Issues in desktop connectivity		3-12
The Ultimate Desk ACCESSory?		3-16
ACCESS.bus hardware released for industrial and commercial environments		3-22
ACCESS.bus controller board connects 125 peripherals to a single PC/AT comm port		3-24
I/O Standard Gains Multivendor Support		3-25
Special Report: ACCESS.bus Specs And Products		3-26
ACCESS.bus: A New Peripheral Bus		3-28

Serial bus looks to standardize connections for peripherals

COMPUTER
DESIGN

COMPUTERS AND SUBSYSTEMS

Serial bus looks to standardize connections for peripherals

Warren Andrews, Senior Editor

In an effort to bring some order to the interconnection of a broad range of accessory devices such as keyboards, locators, bar-code readers and others, Digital Equipment Corp (Maynard, MA) has introduced a new serial connection scheme. In developing the approach, dubbed Access.bus, DEC joined forces with a team from Philips/Signetics (Sunnyvale, CA) using that company's previously developed I²C (inter-integrated circuit) technology. The companies have specified both the physical definition and protocols and are offering them to the public domain with no strings attached.

The Access.bus

As implemented by DEC, Access.bus is designed to handle relatively slow input devices such as keyboards, mice, bar-code readers, modems and some signal transducers for real-time control applications. It's intended to handle up to 14 different devices on a single serial cable which can be as long as 8 m. The bus uses the basic I²C configuration and can handle data rates up to 80 kbits/s (100 kbits/s minus overhead). In addition, the Access.bus cable also carries a +12-V supply voltage for powering each device. According to the specification, the cable carries up to 500 mA.

According to DEC's Paul Nelson, senior engineering manager of input devices, video, image and printer systems group, adding an Access.bus interface to any peripheral device requires only an 8051-family microcontroller with I²C circuitry. "The use of this circuitry should add a maximum of about \$.50/accessory device in OEM quantities," he says. Alternately, he adds, discrete I²C circuitry can be used to implement the interface, but there's

little—if any—advantage. Nelson also hinted that sometime in the future, the basic I²C maximum clock rate might be updated so faster devices such as scanners and/or video and imaging functions could be attached to the bus.

Devices sharing Access.bus are attached with a four-conductor shielded cable connected with a DEC-developed connector (available from both AMP and Molex). The shielding is required to keep the system within FCC radiation requirements. The connector is similar to the standard RJ22 telephone con-

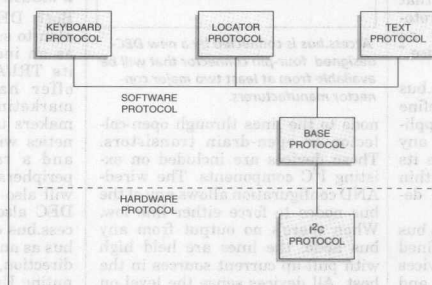
discipline of Access.bus is defined as a subset of the Philips I²C. This is a symmetric, multi-master bus which allows for arbitration among contending masters without losing data. It also provides for cooperative synchronization of the serial clock for exchange of data between bus partners with different maximum clock rates. The scheme allows addressing, framing of data bits and bytes, and byte acknowledgement by the receiver.

Base protocols

The base protocol is common to all types of Access.bus devices and establishes the bus characteristics. The host plays a role as manager of the bus specifying communication between the host and peripheral devices—never between two peripherals. While the I²C protocol allows either a sender or receiver to be the master in a bus transaction, the Access.bus protocol restricts masters to sending and slaves to receiving. The host and all attached devices are, by definition, both master and slave devices and will act as either at the proper time.

The base protocol defines the format of the Access.bus message envelope which is made up of an I²C bus transaction with additional information appended. One item of appended information is a checksum for reliability control. The Access.bus base protocol also specifies a set of seven control and status message types which are used in the

Access bus protocol hierarchy



Access.bus begins with basic I²C protocols and adds a base level common to all devices and other device-specific protocols, which all operate over the same bus.

nect only it's slightly larger to accommodate shielding. The four conductors include the 12-V supply line, a line for data (called SDA for serial data), a line for the clock (labeled SCL for serial clock) and ground. Typical Access.bus devices will have two connectors to allow chaining of devices. "T" connectors, however, may be used with smaller, hand-held devices.

The Access.bus protocol comprises three levels: the I²C protocol, the base protocol and the application protocol. The basic

system configuration.

The configuration process is designed to permit auto addressing and hot-plugging. Auto addressing refers to the way that devices are assigned unique bus addresses in the configuration process without the need for setting jumpers or switches on the devices. Hot plugging lets users disconnect and reconnect devices to the system while it's running without having to reboot the host.

The top level of the Access.bus protocol defines message semantics that are specific to particular

Serial bus looks to standardize connections for peripherals

TECHNOLOGY DIRECTIONS

COMPUTERS AND SUBSYSTEMS

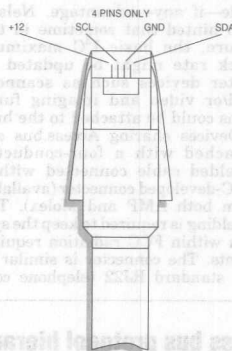
functional types of devices. Different device types have thus far been divided into three classes: keyboards, locators and general text devices. Each class is relatively broadly defined, leaving room for a variety of different devices. The definition for keyboards, for example, allows for the monitoring of a device with up to 255 keys. Locator devices can include not only mice, tablets and trackballs, but also valuator sets (such as dial boxes) with up to 15 valuator and function button boxes with up to 16 buttons. Text devices are defined to include devices providing character streams such as card readers, printers, barcode readers or modems.

In developing the three basic application protocols DEC has attempted to define most of the standard applications. "But there's no restriction on the development of other application protocols as the needs occur. In fact, I anticipate that many additional application protocols will start appearing as Access.bus comes into broader use," says Nelson.

DEC plans to form an Access.bus committee which will help define other standard device-specific application protocols. In addition, any device vendor is free to define its own special device protocol within the general message envelope defined by the base protocol.

At the electrical level, Access.bus functions as Philips initially defined its I²C setup. The host and devices are connected to both the data and clock lines in a "wired-AND" logical configuration. The wired-AND is implemented by connecting the data and clock output stages of each bus

Four pin access bus connector



Access.bus is connected by a new DEC-designed four-pin connector that will be available from at least two major connector manufacturers.

node to the lines through open-collector or open-drain transistors. These devices are included on existing I²C components. The wired-AND configuration allows any of the bus nodes to force either line low. When there's no output from any bus node, the lines are held high with pull-up current sources in the host. All devices sense the level on both the clock and data lines.

When two devices on the bus simultaneously attempt to assert mastership, the logical wired-AND

circuitry takes over. While putting data on the data line, each master independently is sensing the state of that line. Whenever a contending master detects that the state of the data line is different from the data value it's putting out during a clock-high, the contending master backs off and waits for a stop condition before trying again. Thus, two contending masters will both put data on the bus only for as long as they are putting the same data on the data line. The first bit where they differ will cause the contender that put out a "1" (level-high) to back off. For example, two masters trying to send to different bus addresses will resolve the contention by the end of the first byte of the bus transaction. Since the second byte is the address of the master, if both masters are attempting to send to the same address, the contention will be resolved by the end of the second byte.

Industry standard

Both DEC and Philips/Signetics plan to support the Access.bus link as an industry standard. Through its TRI/ADD program, DEC will offer hardware, software and marketing support to peripheral makers using the bus. Philips/Signetics will offer technical support and a range of components for peripheral designers. Each company will also provide development kits. DEC also plans to sponsor an Access.bus consortium to validate the bus as an industry standard. In this direction, the ACE (Advanced Computing Environment), a consortium of 42 major computer makers, has designated Access.bus as an option in the Advanced RISC Computer (ARC) specification.

Note: This article discusses the original DEC implementation of ACCESS.bus which used 12V power on the bus. The ACCESS.bus Industry Group (ABIG) has changed the power requirements for ACCESS.bus to 5V and altered the cable pinout to prevent damage to 12V devices plugged into a 5V system or vice-versa. General ACCESS.bus devices available to the public will use 5V power exclusively, only the original DEC workstation implementation will use 12V. For details, please refer to the ACCESS.bus specification.

ACCESS.bus, an Open Desktop Bus

Peter A. Sichel 1

ACCESS.bus, an Open Desktop Bus

With the recent introduction of the ACCESS.bus product, Digital has affirmed its commitment to open systems and thus to facilitating better solutions for interactive computing. This open desktop bus provides a simple, uniform way to link a desktop computer to as many as 14 low-speed I/O devices such as a keyboard, mouse, tablet, or three-dimensional tracker. ACCESS.bus features a 100-kilobit-per-second maximum data rate, hardware arbitration, dynamic reconfiguration, a mature capabilities grammar to support generic device drivers, and off-the-shelf, low-cost I²C microcontroller technology.

As the cost of personal interactive computing decreases, the range of applications and the need for specialized I/O devices is growing dramatically. Traditional personal computers were designed to accept only a small number of standard devices; adding devices beyond those originally envisioned usually requires specialized hardware or software. Custom interfacing is expensive for vendors and users and thus limits the availability of new devices. ACCESS.bus provides a simple, uniform way to link a desktop computer to a number of low-speed I/O devices such as a keyboard, a mouse, a tablet, or a three-dimensional (3-D) tracker. Designed from the beginning as an open desktop bus, ACCESS.bus facilitates cooperative solutions using equipment from different vendors. This paper describes the ACCESS.bus design and gives some insight into how the idea was adopted at Digital.

Design Goal, Process, and Advantages

The design goal for the desktop bus follows from our experience within the Video, Image and Print Systems (VIPS) Input Device Group with trying to support new devices on Digital terminals and workstations. While various new devices have been successfully prototyped over the years, the need for nonstandard hardware and custom software drivers was always an expensive, time-consuming obstacle. Even after successful prototyping, these devices could not be readily adapted to our standard systems, limiting their use to custom applications. In designing the desktop bus, our goal was to make it as easy as possible to interface previously unavailable I/O devices to our systems in a way that was both practical and marketable. This section explains the benefits of using a desktop bus,

describes the process we went through to convert to a new bus architecture, and summarizes the key advantages of the chosen design.

The basic desktop bus concept is illustrated in Figure 1. The bus allows multiple, low-speed I/O devices to be interconnected and thus interfaced through a single host port. Desktop bus devices such as a keyboard or a tablet, which are not hand-held, provide two connectors and allow another device to be daisy-chained. A hand-held device such as a mouse can be placed at the end of the daisychain, or a connector expansion box can be attached to accommodate additional devices that do not provide two connectors.

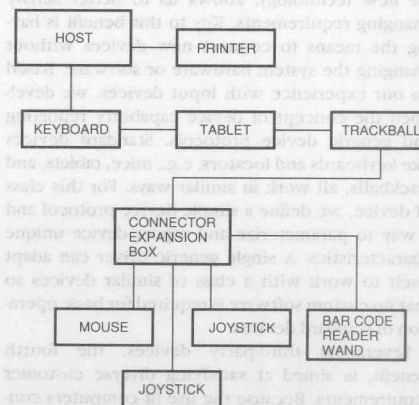


Figure 1 Basic Desktop Bus

The desktop bus has the following benefits:

- Enables greater flexibility and variety of use
- Reduces the cost of connecting multiple devices
- Expedites bringing new technology to market
- Helps leverage third-party devices

The first benefit, greater flexibility, can be simply achieved by allowing additional devices and more modular solutions. We further extended this benefit by designing a way for devices to be added at run time without disrupting system operation. Configuration should be automatic; connecting standard devices should not require powering down or rebooting the system before a new device can be used. The desktop bus supports multiple like devices without switches or jumpers.

The second benefit, reduced cost, was crucial to having the bus accepted as a solution across a wide range of products from low-end video terminals to high-end workstations. We recognized that contemporary electrical techniques could eliminate the need for level translation circuits, -12 volt (V) power supplies, and perhaps some of the protective components used with RS-232 interfacing. Although many devices would now require two connectors, system cost would decrease because we would need to supply only as many connectors as the number of devices to be attached, or possibly one more.

The third benefit, expediting the time to market for new technology, allows us to better satisfy changing requirements. Key to this benefit is having the means to connect new devices without changing the system hardware or software. Based on our experience with input devices, we developed the concept of device capability reporting and generic device protocols. Standard devices like keyboards and locators, e.g., mice, tablets, and trackballs, all work in similar ways. For this class of device, we define a simple device protocol and a way to parameterize and report device unique characteristics. A single generic driver can adapt itself to work with a class of similar devices so that no custom software is required for basic operation of standard devices.

Leveraging third-party devices, the fourth benefit, is aimed at satisfying diverse customer requirements. Because the use of computers continues to proliferate, the range of applications far exceeds that which any one vendor can master.

By making the bus truly open, we encourage third parties to add value to our systems.

The benefits of a desktop bus are significant. But converting to a new architecture, especially one that is not backward compatible, is expensive in terms of the time and effort required. How does a large corporation build agreement to make such an investment decision? The desktop bus project started as a grass roots engineering effort and gradually built momentum. The process was one of dialogue to attract partners. Initially, three groups with slightly different objectives worked together to develop the bus. The visibility of separate groups jointly supporting the bus concept was essential to transform the idea into action. People are more willing to accept an idea that others around them have already adopted.

The three groups that initiated the desktop bus project were our VIPS Input Device Group in Westford, MA, mentioned previously; the Workstation Systems Engineering (WSE) Group, located in Palo Alto, CA; and the Video Advanced Development (A/D) Group in Albuquerque, NM. Our Input Device Group was looking for ways to simplify the process of prototyping specialized input devices and of getting related software support for our video terminals and workstations. WSE was developing a low-cost, personal workstation and needed a flexible way to support multiple input devices without greatly increasing the cost of the base workstation. The Albuquerque A/D Group had been experimenting with next generation I/O devices, i.e., force-feedback joystick, 3-D tracker, and real-time audio and video, and was interested in having these technologies adopted by other Digital groups. This A/D Group had used I²C technology successfully in one of its previous video projects.

In January of 1990, engineers from each group realized they were working on similar problems and began to collaborate. The WSE Group was to build the desktop bus host interface and software drivers into their workstation; the VIPS Group was to help define the device protocols and supply desktop bus keyboards and mice; and the Albuquerque A/D Group was to support bus development and prototype additional devices. Within four months, VIPS had defined the basic protocols and could demonstrate a working I²C keyboard and mouse. These early prototypes helped persuade WSE to support the project and, in turn, helped reinforce the importance of the project to the VIPS Group.

ACCESS.bus, an Open Desktop Bus

Image Processing, Video Terminals, and Printer Technologies

We began presenting the desktop bus idea to interested groups within Digital and received many useful suggestions including

- Use the same keycodes as on the LK201 keyboard to eliminate the need to rewrite keyboard lookup tables.
- Store the country keyboard variation inside the keyboard so users will not need to enter it manually.
- Keep the devices simple, without modes.

In addition, third-party input device vendors made the following suggestions.

- Use a modular connector that is easy to plug and unplug correctly.
- Provide enough power for several additional devices.
- Allow vendors to supply their own device drivers; tuning their own device drivers is part of the value added by the vendor.

The bus idea was elegant and generally well received. Most of the reservations centered around the likely impact on existing system components, the current problems, and whether conversion to the bus was feasible. Because we recognized that other groups were facing tight development schedules, we did not pressure these groups to support our desktop bus work. We presented the desktop bus as a possible solution to interface problems, made our design information available, and worked to incorporate suggestions. But as the development work progressed, more partners supported our effort.

Once we decided to use a desktop bus, we looked at available designs, including the Apple DeskTop Bus, the Musical Instrument Digital Interface (MIDI), and serial buses offered by other semiconductor vendors, and evaluated these alternatives with respect to our design goal. Key advantages of the design chosen, i.e., the ACCESS.bus, are

- Off-the-shelf interintegrated circuit (I²C) microcontroller technology with maximum data rate of 100 kilobits per second (kb/s). This technology is low-cost, yet fast enough for sophisticated input devices like a 3-D tracker.
- Built-in hardware arbitration, which simplifies the software and allows reliable communication without inventing a new protocol.

- Dynamic reconfiguration. The hardware and software allow bus devices to be "hot-plugged" and used immediately, without restarting the system. The devices are recognized automatically and assigned unique addresses. This advantage results in a plug-and-play user interface.

- A mature capabilities grammar to support generic device drivers. An extensible free-form grammar allows devices to describe their characteristics to a generic driver. Most common devices can work with standard drivers.

Bus or network interconnection has become widely accepted as a means of providing flexible open solutions. To appreciate ACCESS.bus, it is helpful to position its performance capabilities with respect to those of other network interconnect technologies, as shown in Table 1.

Table 1 Network Interconnects

Bus Type	Order of Magnitude Performance (kilobits per second)
Apple DeskTop Bus, ACCESS.bus	10-100
LocalTalk	100-1,000
Ethernet	1,000-10,000
FDDI	10,000-100,000

At first glance, the 100-kb/s speed of the ACCESS.bus may seem adequate for large desktop devices like printers and modems. But these devices can transmit long data streams independent of any user activity and, if not restricted, could compromise the interactive performance of the bus. Thus, ACCESS.bus is intended for low-speed activities that people perform with their hands and is fast enough to handle multiple interactive devices like a keyboard, mouse, or 3-D tracker.

Hardware Description

Before discussing the ACCESS.bus design, we present a description of the Philips I²C technology upon which the design is based. Details of the specific ACCESS.bus implementation follow.

Interintegrated Circuit Fundamentals

ACCESS.bus extends the Philips I²C bus to operate off-board and, thus, connect desktop devices. The I²C is a two-wire serial clock and serial data

ACCESS.bus, an Open Desktop Bus

ACCESS.bus, an Open Desktop Bus

ACCESS.bus, an Open Desktop Bus

open-collector bus. An open-collector design means that the clock and data lines are normally in a high-impedance floating state and are pulled up to a logical high state.

A device that wants to send a message waits for any message frame in progress to complete, then asserts a START signal to become bus master and begins to generate data and clock signals. The bus clock is synchronized among all devices by its wired AND connection. Each device, whether transmitting or receiving, stretches the low period of the clock until ready for the next bit to be transferred. When the last device is ready, the bus clock is allowed to go high, generating a rising edge on the serial clock. At this time, all active devices sense the state of the bus data line. For a receiving device, the state represents the received data bit. For a transmitting device, the state determines whether the device has successfully asserted its data on the bus. A transmitter that is sending a logical high state and detects that the data line is being held low by another sender, recognizes that it has lost arbitration and must try again later. When a "collision" or arbitration occurs, no data is lost, one message is transmitted and received, and the remaining messages must be sent again.

I²C data messages are transmitted as 8-bit bytes, with each byte being acknowledged by a ninth ACKNOWLEDGE bit from the receiver. I²C technology also defines unique START and STOP signals to delimit message frames. The first byte of any message frame is always the destination address.

ACCESS.bus Physical Implementation

Details of the physical implementation of ACCESS.bus are as follows:

- Basic electrical configuration. ACCESS.bus uses four-pin, shielded, modular-type connectors that feature positive orientation and locking tabs. Data and power for the bus are transmitted over low-capacitance, four-wire, shielded cable. The four conductors are used for ground, serial data, serial clock, and +12 V.
- Available power. The maximum available power for all devices is 12 V at 500 milliamperes (mA). ACCESS.bus devices may supply their own power from a separate source, if needed. A power-up reset circuit must still be provided to reset the device when bus power is applied.
- Cable length. The maximum cable length for the entire bus is 8 meters. The limiting factor is a

maximum capacitance not to exceed 700 picofarads (pF).

- Number of devices. The maximum number of ACCESS.bus devices allowed on the bus is 14. Limiting factors are the device addressing range and the power distribution (a total of 500 mA for all devices).
- Hardware interfaces. ACCESS.bus hardware interfaces are implemented using standard I²C microcontrollers developed by the Signetics Company, or under license from Philips Corporation. (Signetics Company is a division of North American Philips Corporation.)

ACCESS.bus Protocol

Every device on the bus is a microcontroller with an I²C interface and behaves as either a master transmitter or a slave receiver, exclusively, as defined by the I²C Bus Specification.

Message Format

A message transmits information between a device and the computer or between the computer and one or more devices. There is one exception: a device may attempt to reset other devices assigned to the same address by sending a Reset message to itself.

ACCESS.bus messages have the following format:

Byte Number	Bit Number	
	[1 2 3 4 5 6 7 8]	
1	[destaddr 0]	Destination address
2	[srcaddr 0]	Source address
3	[P length]	Protocol flag, length (the number of data bytes from 0 to 127)
4 through (length + 3)	[body]	Consists of 0 to 127 data bytes
length + 4	[checksum]	

Initially, devices respond to a default power-up address. During the configuration process, the computer assigns a unique address to every device on the bus. Messages are either device data stream (P=0) or control/status (P=1), as indicated by the

ACCESS.bus, an Open Desktop Bus

ACCESS.bus, an Open Desktop Bus

Image Processing, Video Terminals, and Printer Technologies

protocol flag. The minimum length of a message is 4 bytes; the maximum length is 131 bytes (127 data bytes and 4 bytes for overhead). The message checksum is computed as the logical XOR of all previous bytes, including the message address.

Standard Messages

The ACCESS.bus protocol defines the seven standard interface messages summarized in Table 2. Parameters defined within the body of the message are listed in parentheses.

Identification

Since the ACCESS.bus is a bus-topology network, unique identification strings are used to distinguish devices. These strings are structured as follows:

protocol revision:	1 byte (e.g., "A")
module revision:	7 bytes (e.g., "X1.3 ")
vendor name:	8 bytes (e.g., "DEC ")
module name:	8 bytes (e.g., "LK501 ")
device number:	32-bit signed integer

The module revision, vendor name, and module name strings are left-justified ASCII character strings padded with spaces. The device number string is a 32-bit two's complement signed integer and may be either a random number (if negative) or a unique serial number (if positive).

Configuration Process

The configuration process is used to detect what devices are present on the bus, assign each device a

unique address, and connect devices to the appropriate software driver. Configuration normally occurs at system start-up, or at any time when the computer detects the addition or removal of a device.

Power-up/Reset Phase

When reset or powered up, a device always reverts to the default address and sends an Attention message to alert the computer to its presence. At system start-up or reinitialization, the computer sends a Reset message to all I²C addresses in the ACCESS.bus device address range (14 messages) to ensure that all devices on the bus respond at the power-up default address.

Identification Phase

To begin address assignment, the computer sends an Identification message at the device default address. Every device at this address must then respond with an Identification Reply message. As each device sends its message, the I²C arbitration mechanism automatically separates the messages based on the identification strings. The computer can then assign an address to each device by including the matching identification string in the Assign Address message. When a device receives this message and finds a complete match with the identification string, it moves its device address to the new assigned value. As soon as a device has a unique address, it is allowed to send data to the computer.

The I²C physical bus protocol allows multiple devices on the bus at the same time if those devices

Table 2 Standard ACCESS.bus Protocol Messages

Computer-to-device Messages	Purpose
Reset ()	Force device to power-up state and default I ² C address.
Identification Request ()	Ask device for its "identification string."
Assign Address (identification string, new address)	Tell device with matching "identification string" to change its address to "new address."
Capabilities Request (offset)	Ask device to send the fragment of its capabilities information that starts at "offset."
Device-to-computer Messages	
Attention (status)	Inform computer that a device has finished its power-up/reset test and needs to be configured; "status" is the test result.
Identification Reply (identification string)	Reply to Identification Request with device's unique "identification string."
Capabilities Reply (offset, data fragment)	Reply to Capabilities Request with "data fragment," a fragment of the device's capabilities string; the computer uses "offset" to reassemble fragments.

ACCESS.bus, an Open Desktop Bus

ACCESS.bus, an Open Desktop Bus

are transmitting exactly the same message. In the rare event that two like devices report the same random number or are mistakenly assigned to the same address, each interactive device transmits a Reset message to its assigned address prior to sending its first data message after being assigned a new address. The self-addressed Reset message forces other devices at the same address back to the power-up default address, as if they had just been hot-plugged. The message guarantees that each device has a unique address, but not until the device is actually used. The pseudo-random number (or serial number, if available) distinguishes devices at identification time before they are used, allowing the host to inventory which devices are present.

Capabilities Phase

Device capabilities is the set of information that describes the functional characteristics of an ACCESS.bus peripheral. The purpose of capabilities information is to allow software to recognize and use the features of bus devices without prior knowledge of their particular implementation. By having locator devices report their resolution, for example, generic software can be written to support a range of device resolutions. Capabilities information provides a level of device independence and modularity.

The structure of capabilities information is designed to be simple and compact for efficiency, but also extensible to support new devices without requiring changes to existing software or peripherals. These objectives are supported by making the structure hierarchical and representing capabilities information in a form that applications (and humans) can use directly. The capabilities information is an ASCII string constructed from a simple, readable grammar. The grammar allows text strings to be formed into lists, nested lists, and lists with tagged elements. The capabilities string for a locator might read as follows:

```
(prot(locator)
 type(mouse)
 buttons( 1(L) 2(R) 3(M) )
 dim(2) rel res(200 inch) range(-127 127)
 d0(dname(X))
 d1(dname(Y))
)
```

After assigning a unique address to a device, the computer retrieves the device's capabilities string as a series of fragments using the Capabilities Request and Capabilities Reply messages. The com-

puter then parses the capabilities string to choose the appropriate application driver for the device. The parsed string is also made available to application programs using the device.

Normal Operation

During normal operation, the computer periodically requests inactive devices to identify themselves. If a device is found to be missing, or a new device appears by sending an Attention message at the default address, the computer sends an Identification Request message to each device address previously recorded as in use (up to 14 messages) to confirm which devices are still present. The computer also sends a Reset message to each device address previously recorded as not in use. The computer then begins the address assignment process by sending an Identification message to the default address and assigning each device that responds to an unused device address.

Generic Device Concepts

ACCESS.bus uses the concept of generic device drivers to support familiar I/O devices using only a few drivers. Generic specifications for keyboards, locators, and text devices have been developed.

Keyboards

The keyboard device protocol attempts to define the simplest set of functions from which a Digital LK201 or a common personal computer keyboard user interface can be built. A generic keyboard consists of an array of key stations assigned numbers between 8 and 255. When any key station transitions between open and closed, the entire list of key stations currently closed or depressed is transmitted to the host. This reporting scheme is functionally complete; the host can detect every key transition, and the scheme provides information about the full state of the keyboard on each report. No special resynchronization reports are required.

In addition to reporting key stations, the generic keyboard device can support simple feedback mechanisms such as keyclicks, bells, and light-emitting diodes. These mechanisms are controlled explicitly from the host so that minimal keyboard state modeling is required. The capabilities information is used to identify the keyboard mapping table and the feedback mechanisms available. The keyboard mapping table can also be stored in the keyboard itself as part of the capabilities string.

ACCESS.bus, an Open Desktop Bus

Image Processing, Video Terminals, and Printer Technologies

Locators

The locator device protocol is designed to accommodate a range of basic locator devices such as a mouse or tablet. More complex devices can be modeled as a combination of basic devices or can provide their own device driver, thus minimizing the burden on the protocol.

A generic locator consists of one or more dimensions described by numeric values and, optionally, a small number of key switches. The standard driver requires the locator device to identify the type of data it will report from a small list of options and adjusts to handle this data type. These options are

- Number of dimensions, e.g., two, for a mouse or a tablet
- Dimension type: absolute, i.e., referenced to some fixed origin, like a tablet; or relative, i.e., changed since last report, like a mouse
- Resolution in divisions per unit, e.g., counts per inch or counts per revolution
- Dynamic range of values that can be reported, i.e., the minimum and maximum values
- Number of key switches, from 0 to 15

The assignment of scalar-value dimensions returned from one or more devices to the user interface functions is left to the application. However, to accommodate most conventions, the scalar dimensions and the key switches can be labeled in the capabilities string.

Text Devices

The text device protocol is intended to provide a simple way to transmit character data to and from character devices such as a bar code reader or a small character display. A generic text device transmits a stream of 8-bit bytes from a character set. Simple control messages are defined to support flow control and to select communication parameters that might be used to interface with a modem. The capabilities string contains information that identifies the specific character set and communication parameters used.

Summary

The ACCESS.bus network interconnect offers the possibility of a standardized, low-speed, plug-and-play serial communications channel that can untangle peripheral interfacing and open the way to new

applications. As the advantages of this open desktop bus design become well known, we expect wider adoption of this product. The ACCESS.bus is currently implemented on Digital's Personal DECstation 5000 workstation, with implementations underway for the next generation of RISC workstations and video terminals.

Acknowledgments

Many people contributed to the design and development of ACCESS.bus. I would especially like to acknowledge Tom Stockebrand and Tom Furlong for their vision and early support; Chris Cued, Mark Shepard, and Ernie Souliere for their contributions to the ACCESS.bus electrical design and protocols; and Robert Clemens for the excellent demonstration hardware and firmware development support.

General References

D. Lieberman, "Desktop Bus Is Born Free," *Electronic Engineering Times* (September 2, 1991): 16.

ACCESS.bus Developer's Kit (Palo Alto, CA: Digital Equipment Corporation, Workstation Systems Engineering TRI/ADD Program, 1991).

Signetics I²C Bus Specification (Sunnyvale, CA: Signetics Company, a Division of North American Philips Corporation, February 1987).

Reprinted with permission from the **Digital Technical Journal**. Copyright © 1991 Digital Press/Digital Equipment Corporation, One Burlington Woods Drive, Burlington, MA 01830.

Issues in desktop connectivity

Authors: Ata Khan and Greg Goodhue, Sunnyvale

INTRODUCTION

Desktop connectivity has become a major issue in system design. This paper identifies the criteria for evaluating a desktop bus or interconnect from data transfer rates and protocol flexibility to ease of implementation. It also compares the performance and cost implications of the typical desktop connectivity alternative.

DESKTOP CONNECTIVITY

Desktop connectivity is a method of connecting computer peripherals that are most often found on work surfaces, or desktops, to a host computer or workstation. Most often, desktop peripherals are low-speed Input/Output devices such as keyboards, mice, tablets, joysticks, and modems. This paper discusses devices which are usually dedicated to one computer, such as a PC, rather than high-speed peripherals such as laser printers and disk drives that are shared between users.

INTEREST IN DESKTOP CONNECTIVITY

Current interest in desktop connectivity comes from both users and manufacturers of PCs and workstations.

From the User's Side

Connecting low-speed I/O devices such as keyboards, mice, tablets, modems, and low-speed printers to a PC or a workstation has resulted in two problems for the user: a shortage of ports, and an excess of cabling.

Shortage of ports is a big problem, especially with computers that have few slots or none at all. A computer with two serial ports, a parallel port, and a keyboard port is soon overwhelmed by peripherals requiring external expanders or switches of some sort. In the worst case, the user may be unable to use all the available peripherals at one time. Newer notebook computers exacerbate this problem by having fewer ports than desktop machines.

An excess of cabling and connectors is one of the bane of connecting several desktop peripherals. By the time a user has figured out all the right cables, connectors, and gender changers, there is usually a small jungle on the desktop. From the user's point of view, there must be a better way.

From the Manufacturer's Side

Providing all the ports to which low-speed desktop I/O devices may be connected is expensive. Besides the actual hardware

involved, valuable motherboard real estate is used and, especially in compact machines such as notebooks or pen-based computers, it is tricky to provide a lot of external connectors in very limited space. Also, the number of ports provided is usually either too few or too many, usually the former.

Providing ports in hardware means connecting them to the CPU on the motherboard. Usually, this means each port must be provided with its own interrupt and dedicated address range. With systems, such as IBM type PCs, already being under severe constraints with regard to the number of I/O interrupts, adding sufficient ports for desktop use may become a problem.

The CPU is the de facto manager of all these low-speed desktop peripherals; this means that signals from these slow devices are now moving on the same channels that high-speed devices such as disks are connected. Allowing bicycles on the *Autobahn* is a good analogy to the current system.

Since each device has its own port, it also has its own device driver that "connects" the application software to the peripheral. This means that device drivers are hardware specific and different for each peripheral connected.

Reconfiguring devices without powering down a system is a major advantage for both manufacturers and users, as anybody who has ever powered off a networked UNIX workstation knows. If it were possible to add and remove peripherals dynamically, such as different keyboards or tablets, without reconfiguring or re-booting the system, this would be a major operational advantage.

CRITERIA FOR SELECTING A DESKTOP BUS

Ideal criteria for selecting a desktop bus are outlined below:

- Low-cost: Off-the-shelf, commodity type components at a minimum—ideally, just one—of these.
- Daisy-chained: By allowing components to connect into each other, both of the problems the user has with a shortage of ports and an excess of cables are solved.
- Dynamic reconfiguration: How plugging and un-plugging of devices should be allowed without having to power down the system or re-boot it. This allows peripherals to be added, removed or defective ones swapped without long delays and inconvenience.

- Uniform interfaces: With daisy-chained devices, the hardware interface is uniform by definition. This allows the same hardware layer drivers to be used for all peripherals. Software uniformity is also ideal but harder to implement and can, to some extent, be achieved through a layered protocol.
- Sufficient bandwidth: While devices such as keyboards, mice, and tablets usually do not present much of a problem, devices such as modems and printers may. System simulation should be carried out to see that the chosen bandwidth meets these needs without causing excessive delays, lost data, or other problems.
- Sufficient capacity: At the very minimum, six devices should be allowed (keyboard, mouse, tablet, modem, printer, misc.) and the cable specification should support this minimum.
- Inexpensive peripherals: Peripherals should be available that support this standard and are not significantly more expensive, if at all, than those available for other standards.
- Open standard: There should be no royalties, patent issues, or licensing fees associated with the use of a desktop connectivity standard.

CURRENT IMPLEMENTATIONS OF DESKTOP CONNECTIVITY

Three major platforms implement desktop connectivity standards: IBM and compatible PCs, workstations, and Macintoshes.

The PC platform and workstation platforms use dedicated I/O ports (RS-232 or other) for each peripheral being connected. Thus, for an IBM-style PC, each desktop peripheral has to have a port on the machine: one for the keyboard, one for the mouse, one for the tablet, etc. This gives rise to an onerous proliferation of cabling, connectors, and driver software where space is at a premium. In IBM-style notebook computers, there is usually a shortage of ports caused by limitations of space as well as power. The current solution is cumbersome, expensive to implement, and inelegant.

Workstations use basically the same approach to desktop connectivity. However, there is some saving grace here that restrictions on space, power, and cost are less severe than PCs.

The Macintosh™ solution, known as the Apple Desktop Bus™, is better than the PC and workstations solutions in the sense that it is more efficient in its use of space since peripherals can be daisy-chained and peripheral does not require a dedicated port.

Issues in desktop connectivity

ACCESS.bus™: A PROPOSED DESKTOP CONNECTIVITY STANDARD

A desktop connectivity system for interconnecting low-speed peripherals was originally developed by Digital Equipment Corporation (Digital) in partnership with

Philips Semiconductors and offered as an open standard. Called ACCESS.bus (a bus for connecting ACCESSory devices to a host system), the standard embodies most of the criteria of an ideal desktop connectivity standard.

ACCESS.bus (also referred to as A.b) is a

daisy-chained bus (see Figure 1 which allows up to fourteen devices (there are provisions for these devices themselves controlling other devices for expansion.) The total length of the cable is allowed to be eight meters and the data throughput rate of this bus is approximately 80 kbits/sec.

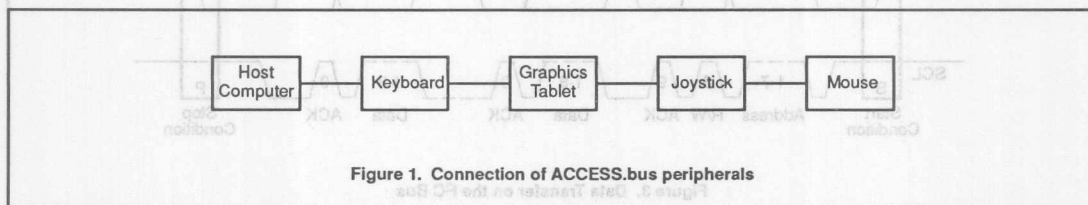


Figure 1. Connection of ACCESS.bus peripherals

The ACCESS.bus specification, technology, and trademarks are now owned and offered by the ACCESS.bus Industry Group (ABIG). ABIG is an open independent association,

organized to maintain and promote ACCESS.bus as an "Open Industry Standard."

ACCESS.bus Structure

A.b is a layered protocol supporting a daisy-chained bus topology. There are three layers in the protocol (see Figure 2):

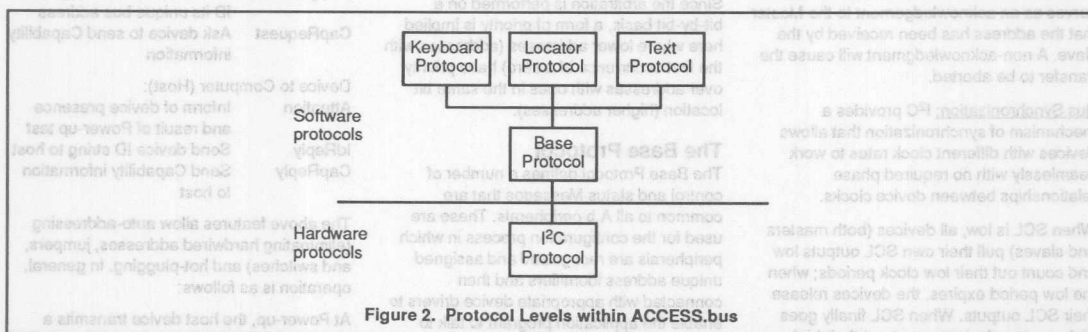


Figure 2. Protocol Levels within ACCESS.bus

1. The Hardware (I²C) Protocol Layer: This layer is based on the Inter-Integrated Circuit, or I²C, serial protocol developed by Philips. This protocol defines a scheme for performing bus transactions with addressing, framing of bits into bytes, and acknowledgement of each byte by the receiver.
2. The Base Protocol Layer: This level is common to all A.b devices and builds on the Hardware layer to establish an asymmetric interconnect between a host computer and a number of peripheral devices. The A.b message envelope with control and status messages is defined here.
3. The Application Protocol Layer: In this layer, devices are differentiated with message semantics that are specific to particular kinds and classes of devices.

In short, the mailman is I²C, the mail envelope is the Base Protocol, and the contents of the message are the Application Protocol.

The Hardware Protocol (I²C)

The Philips Inter-Integrated Circuit protocol, or I²C, is a 2-wire (clock and data) serial protocol which allows wire-AND connection of devices to the clock and data lines. The protocol allows devices to be either masters or slaves at any given bus transaction time (masters control the transaction and generate the clock signal).

I²C is a symmetric multimaster bus where several masters can contend for the bus and an arbitration scheme resolves bus mastership without loss of data or re-transmission. Different clock rates are allowed on the bus with a cooperative synchronization scheme for the serial clock;

this allows bus transactions to be performed between bus devices with different clock rates and without requiring any clock locking schemes.

Device Connection: All devices are wire-AND'ed identically to the clock (SCL) and data (SDA) lines. Outputs must be open-drain or open-collector. Thus, any device may pull the bus low. When neither line is pulled low, the lines are held high by pull-up resistors. Every device must be able to sense the state of the line. In practice, this means a common ground level is required so that all devices see more or less the same input threshold.

Bus Transaction: The bus is idle when both SDA and SCL lines are high after a transaction has been completed or after power-up. To initiate a transaction, a master device pulls the SDA line low. This represents a Start condition on the bus and all devices

are required to sense its occurrence. Once the Start condition is asserted, the master then takes the SCL line low, and starts

putting data on the SDA line by driving it low or leaving it high and pulsing the SCL line high and then back low. Data (SDA) is not

allowed to change while the clock (SCL) line is high (see Figure 3).

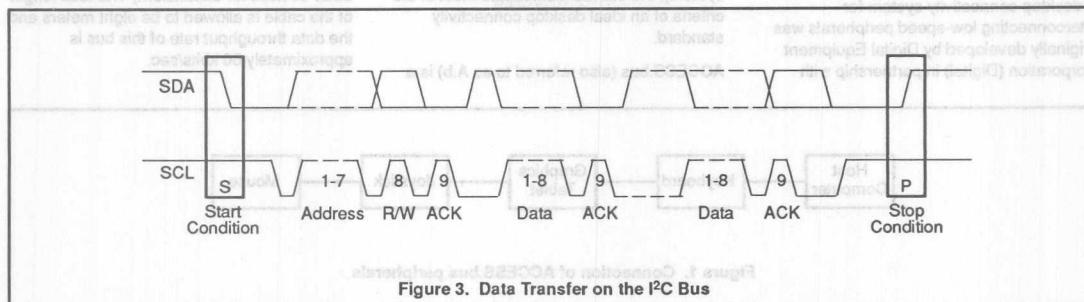


Figure 3. Data Transfer on the I2C Bus

The master outputs the Address of the slave as the first byte transmitted (MSB first). After 8 bits have been transmitted, the slave, if one exists and recognizes its address, will pull the Data line low on the 9th clock bit time. This serves as an acknowledgement to the Master that the address has been received by the slave. A non-acknowledgment will cause the transfer to be aborted.

Bus Synchronization: I2C provides a mechanism of synchronization that allows devices with different clock rates to work seamlessly with no required phase relationships between device clocks.

When SCL is low, all devices (both masters and slaves) pull their own SCL outputs low and count out their low clock periods; when the low period expires, the devices release their SCL outputs. When SCL finally goes high, master devices count out their high clock periods until the master with the shortest clock period pulls SCL low again. Thus, the SCL low period is defined by the device with the longest low period and its high period by the device with the shortest high period.

This cooperative synchronization allows all devices to use a common clock and allows slower peripherals to regulate the speed of the bus.

Multimaster Arbitration: If two or more masters start transmitting on the bus, given that the clock is synchronized, each master then examines the state of SDA to see if its state is the same as the value of the bit it transmitted. The instant a master sees a difference, it knows it has lost arbitration and terminates its attempted transaction. Thus, arbitration between masters sending out different bus address values will always be uniquely resolved within the address transmission time. If masters send exactly the

same message with the same contents to the same address, then the message is sent but, in cases where this is not desirable, software interlocks can prevent this situation.

Since the arbitration is performed on a bit-by-bit basis, a form of priority is implied here where lower addresses (addresses with the first occurrence of a zero) have priority over addresses with ones in the same bit location (higher addresses).

The Base Protocol

The Base Protocol defines a number of control and status Messages that are common to all A.b peripherals. These are used for the configuration process in which peripherals are recognized and assigned unique address identifiers and then connected with appropriate device drivers to enable the application program to talk to them.

A Message has five parts to it:

- The first byte is the address of the destination or the receiver.
- The second byte is the address of the source or the transmitter.
- The third byte specifies whether the body of the message is control or data, if there are any sub-devices (0 to 3), and the length of the message body following in bytes.
- This part is the Message body, from 1 to 32 bytes.
- A Checksum. This byte is the bit-wise XOR of all the preceding bytes in the message.

While the detailed syntax and structure of the messages will not be discussed here, there are seven basic Messages defined as follows:

Computer (Host) to Device:

Reset	Force device to Power-up state and default address
IdRequest	Ask device for its identification string (ID)
AssignAddress	Give device with recognized ID its unique bus address
CapRequest	Ask device to send Capability information
Device to Computer (Host):	
Attention	Inform of device presence and result of Power-up test
IdReply	Send device ID string to host
CapReply	Send Capability information to host

The above features allow auto-addressing (eliminating hardwired addresses, jumpers, and switches) and hot-plugging. In general, operation is as follows:

At Power-up, the host device transmits a general Reset to all devices (assigned the same default address at power-up). The devices initialize themselves, perform a self-test and send the result back to the host by an Attention message. On receiving an Attention message, the host sends an IDRequest message to the Default Address, each device at this address sends a unique 28-byte ID string back to the host; the IDReply. On getting the IDReply, the host is then able to assign a unique A.b address to the device.

A Hot-plugged new device will send an Attention Message to the host seeking to be assigned an address. The host periodically checks the last logged configuration by sending IDRequest messages to all inactive devices. These actions allow the bus configuration to be dynamically altered.

In this layer, the particular peripheral's device driver has to be found, loaded and connected to the application program. This is done by determining the device type and Capability

Issues in desktop connectivity

Information by the CapRequest/CapReply messages.

The Application Protocol

Application level protocols are device-specific and the message semantics are different for different defined device types as well as different sub-types. While device drivers are different at this level, the Hardware and Base layers are independent of the device type allowing much firmware to be shared. Even at the Application Protocol level, a common definition structure is used for similar device types to encourage a common approach to writing A.b device drivers.

So far, devices are classified into three broad types:

- i. Keyboards: Up to 255-key devices are supported. Special function keys and annunciator support is built-in.
- ii. Locator devices: Intended for Mice, tablets, etc. This provides for devices with up to 15 degrees of freedom and up to 16 binary keys.
- iii. Text devices: These are defined as data stream devices such as printers, modems, etc.

Details of the semantics are in the ACCESS.bus specification. Flow Control (XON/XOFF) is provided with a character count being specifiable.

ACCESS.bus IMPLEMENTATION

The A.b controller must support all layers of the protocol for both the system and peripheral ends of the bus, the I²C hardware, the A.b Base layer, and the A.b Application layer. This is most efficiently done by using a microcontroller with an I²C interface with enough on-chip memory to implement the A.b firmware. Thus implementing the entire protocol in one, off-the-shelf, commodity priced component.

Philips manufactures a broad line of 80C51-based microcontrollers with built-in I²C interfaces and varying amounts of on-chip memory ranging from 2k bytes to 32k bytes of program memory. EPROM and OTP versions are available for system

development as well as production. These microcontrollers provide all the intelligence for managing device communications for all 3 layers of the protocol. Of course, since the Base and I²C layers are common, this firmware may be reused. Even at the Application layer, the common message structure allows some reusability.

Since the 80C51 is an industry standard for microcontrollers, there is an abundance of third party support in the form of assemblers, compilers, development systems with In-Circuit-Emulation and symbolic debugging capability, EPROM programmers, etc. These tools are usually low-cost and are almost always PC-based.

The physical connection itself is made by using a 4-pin connector that Molex and AMP will offer as a standard catalog item. The 4 wires are Power, Ground, SDA, and SCL. The shielded cable used has roughly 70 pF/meter of capacitance and up to 8 meters can be used.

ACCESS.bus VIS-A-VIS THE IDEAL AND VERSUS AN EXISTING STANDARD

Comparing A.b to the list of ideal desktop connectivity attributes, we see that it fulfills them all except for the availability of low-cost off-the-shelf peripherals.

A.b uses low-cost standard components, supports dynamic reconfiguration, is daisy-chained, and allows fourteen devices (and more via sub-devices). With a data rate of 80kbits/second, it is ample for the class of low-speed desktop peripherals for which it is intended. It presents a uniform hardware and software interface allowing re-usability of most firmware and has built-in error checking at both byte and message levels.

ACCESS.bus will be implemented in forthcoming Digital workstations and, working with the ACE initiative, Digital and Philips will be striving to make it a standard for desktop connectivity. The issue of availability of peripherals presents a circular argument (which comes first) but, at this point, a keyboard, a mouse, and a graphic tablet are

being developed for ACCESS.bus by major peripheral manufacturers. ACCESS.bus connectors are being developed by Molex and AMP and will be a standard item in their offerings.

The closest existing alternative to ACCESS.bus is Apple Computer's ADB (Apple Desktop Bus, see Table 1). ADB is a daisy-chained bus but has the following somewhat severe limitations versus ACCESS.bus:

- ADB does not support hot-plugging or dynamic reconfiguration.
- ADB has a maximum data rate of 10kbits/sec versus A.b's 80kbits/sec.
- ADB has a 3 device limit versus 14 for A.b (and A.b can support more via sub-devices).
- ADB is a closed, proprietary bus, whereas A.b is fully open.
- ADB is specified to work over a 5 meter length, while A.b can go 8 meters.

Table 1. ACCESS.bus vs. Apple Desktop Bus

Feature	Apple Desktop Bus	ACCESS.bus
Hot plugging	Limited	Full support
Transfer rate	10kbit/sec	80kbit/sec
Maximum number of devices	3 devices	14 devices
Maximum length	5 meters	8 meters
Availability	Proprietary	Open to all

CONCLUSION

The issue of desktop connectivity for PCs and workstations has been neglected and solved piecemeal so far. ACCESS.bus represents the first coherent, well-defined, desktop connectivity standard which is solidly backed and, based on what it offers both users and manufacturers, should become established as a market standard as well.

The Ultimate Desk ACCESSory?

Issues in desktop connectivity

CIRCUIT CELLAR **I N K**®

THE COMPUTER APPLICATIONS JOURNAL

SIGNAL PROCESSING

SPECIAL SECTION:

Embedded Interfacing

RISC vs. DSP

The Photonic Transistor
Instant PC Boards

August/September, 1992 — Issue #28



\$3.95 U.S.
\$4.95 Canada

The Ultimate Desk ACCESSory?

The
Ultimate
Desk
ACCESSory?SILICON
UPDATE

Tom Cantrell

a

s a longtime user of both Macs and PCs, I'm uniquely qualified to throw in my two cents regarding the relative strengths and weaknesses of each.

With my Mac hat on, I amuse myself by watching Microsoft rake in the dough with constant "upgrades" of Windows (e.g., 286, 3.0, 3.1, NT?), which millions of users seem to accept with good grace. Me? I won't consider anything sooner than NT and probably at least version 3+ at that.

Meanwhile, as a PC proponent, I point out the plethora of engineering software that's available and the incredible price-to-performance comparison of clones. When I have to develop embedded hardware and software, the PC is king.

One area where the Mac has a decided advantage is built-in networking. The most well-known example is AppleTalk, which provides as-painless-as-it-gets networking of Macs and laser printers. More recently, the Mac also features the Apple Desktop Bus (ADB), which provides a low-cost, user-friendly way to daisy chain a variety of desktop input devices. For example, setting up a system with a keyboard, mouse, trackball, and digitizer is easy.

By contrast, figuring the same setup on the PC is an ugly job involving a strange variety of boards (each with the dreaded DIP switches, of course), connectors, and cables. The latter wind their way in an inevitably tangled mess from desktop to the PC.

If technology is developing at such a great rate, why am I still driven to my knees fumbling in that forbidding rat's nest of cables that lurks behind

my PC? Presumably, "wireless" technology will eliminate cable clutter someday, but until then, there's got to be a better way.

ACCESS.bus TO THE RESCUE

In an effort to bring ADB-like sanity to the PC world, DEC and Philips/Signetics are proposing the ACCESS.bus. Like the ADB on the Mac, ACCESS.bus is a simple, low-cost daisy-chain bus (see Figure 1). However, with the benefit of hindsight, ACCESS.bus offers a number of improvements compared to ADB including

- **High-Speed Data Transfer**—ACCESS.bus is much faster at 100K bits per second versus ADB 10K bits per second. This speed may seem like overkill for a keyboard, but it does allow for plenty of expansion without performance problems. Also, a fast version of ACCESS.bus—400K bits per second—will be offered later.

- **Larger Number of Devices**—Although ADB theoretically supports 16 devices, Apple documentation states "performance will probably deteriorate if more than three devices are daisy chained" (!!). Also, ADB limits the power delivered to devices to a total of 500 mA. Meanwhile, thanks to a higher bandwidth, ACCESS.bus can support 14 devices, and no upper limit is imposed on total device power consumption.

- **Longer Cable**—More devices need more cable, so ACCESS.bus stretches the limit from the 5-meter ADB limit to 8 meters. This amount seems quite adequate for desktop work, but if it is not, active "repeaters" can be used.

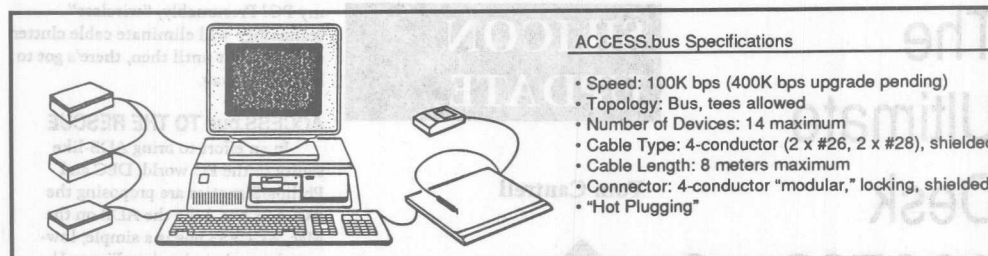
- **"Hot Plugging"**—Apple specifically warns against the practice of adding a device to an active ADB bus. The ability of ACCESS.bus to let something plug in at any time could support unique identification and security applications, such as an ACCESS.bus/EEPROM-based "key."

Testifying to the credibility of the new bus are connector leaders Molex and Amp, who provide the neat ACCESS.bus phonelike modular connector (see Figure 2). It's a little bigger than a phone connector and

If DEC and Philips/Signetics have anything to say about future PCs, your days of crawling around under your desk routing cables may soon come to an end.



The Ultimate Desk ACCESSory?



ACCESS.bus Specifications

- Speed: 100K bps (400K bps upgrade pending)
- Topology: Bus, tees allowed
- Number of Devices: 14 maximum
- Cable Type: 4-conductor (2 x #26, 2 x #28), shielded
- Cable Length: 8 meters maximum
- Connector: 4-conductor "modular," locking, shielded
- "Hot Plugging"

Figure 1—ACCESS.bus aims to eliminate the clutter of incompatible cables running out the back of your PC by using a "desktop bus" scheme similar to that used by today's Macintosh computers.

shielded to minimize RF problems. The specification calls for four-conductor cable (also shielded) with heavier wire for +5 volts and ground (#26 vs. #28). Unlike the DIN connectors used by ADB and current PC keyboards, the modular ACCESS.bus connector has the advantage of easy orientation. I don't know about you, but I inevitably end up "spinning" DIN connectors a lot even when I can see what I'm doing, not to mention when I'm groping through the all too typical "blind insertion."

Another plus is positive locking. Unlike a phone connector, the dual-release ACCESS.bus connectors seem to make "getting a grip" easier. Notice

how the entire connector is streamlined, easing wire routing and minimizing back panel clutter. The design also minimizes the "fishhook" syndrome exhibited by existing connectors (e.g., those DB-25s with the long knurled screws), which seem to get hung up on every possible obstacle as if they were possessed by some mystical attraction.

PC THE LIGHT

Besides causing grief for users, a multitude of desktop interfaces caused problems for DEC keyboard manufacturing. They had to offer *X* distinct keyboards, where *X* equals the number of popular layouts multiplied by the number of different interfaces. Thus, the seeds were sown for ACCESS.bus.

DEC approached Apple to see if they would consider offering ADB as an open standard. Apparently, Apple's response was something along the lines of "Hey, great idea—not!" so DEC started casting around for alternatives.

Here's where Philips/Signetics enters the picture. To make a long story short, the resulting ACCESS.bus is simply a derivative of that company's Inter-Integrated Circuit (I²C) bus.

I²C was originally designed as kind of a "LAN-in-a-Box," allowing easy connection between processors and interface chips without the bulk and expense of a full-speed parallel bus. Though you may not be familiar with it, I²C is arguably the world's leading LAN because the bus is widely used in high-volume consumer electronics, such as TVs, stereos, and phones. Meanwhile, Philips/Signetics (and others under license) offer a plentiful variety of I²C add-on chips including micros, EEPROMs, real-time clocks, ADC, DAC, and so forth.

I²C is surprisingly sophisticated, despite its low chip cost, simple wiring, and a simple clocked serial port basis where data (SDA) is sampled when the clock (SCL) is high. On top of the basic communication mechanism, I²C layers a message format consisting of the destination address, a read/write flag, and the data framed by start and stop conditions. Furthermore, each byte transferred requires an ACK

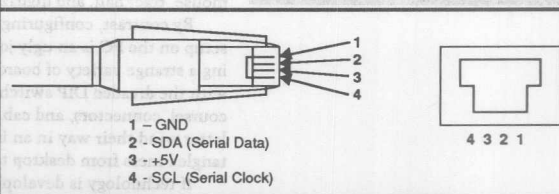
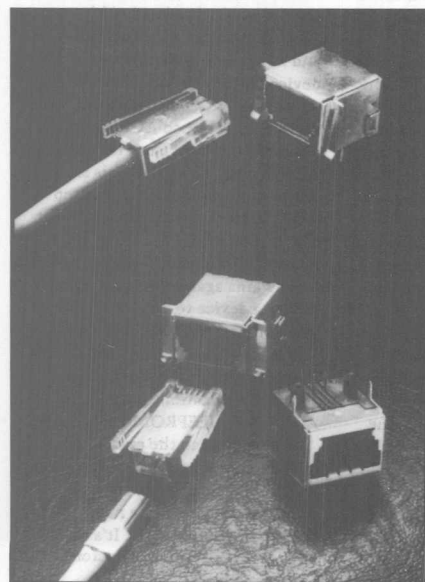


Figure 2—Based on I²C, ACCESS.bus uses a simple four-wire interface that provides not only a data channel, but also power to peripherals. The proposed modular connector locks in place and eliminates orientation confusion.

The Ultimate Desk ACCESSory?

or NAK from the recipient (see Figure 3).

PC is smart when generating the clock and when dealing with varying speed devices in particular. Relying on the use of open-collector drivers, slow devices can request the equivalent of wait states by holding the clock low. Because the clock synchronization is automatically handled by the PC hardware, you don't have to change DIP switches or software settings if you add a slow device.

As a multimaster bus, PC must face arbitrating between simultaneous data transfers. Most serial networks adopt a variant of "collision detection" in which each potential master monitors its own transmission and everyone backs off if a collision (i.e., what's on the wire isn't what was sent) is detected. Once again, relying on the open-collector nature of the bus, PC adopts an interesting variation where at least one of the messages will get through. During a simultaneous data transfer, all masters continue to output as long as the data on the wire matches what they are sending. Eventually, a particular master will output a high, but the output from another master will be holding the wire low. At that point, the loser (the master with the high output) detects a collision, quits transmitting, and has to try again. The process continues until a single winner, one whose message gets through, remains.

ACCESS.bus makes a few changes to the PC protocol. First, of the 128 addresses (7-bit address) defined by PC, 16 are allocated for ACCESS.bus devices as follows:

- 50H: Host computer
- 6EH: Default power-up address
- 52-6CH (even addresses): 14 assignable device addresses

An ACCESS.bus message superimposes more information on top of the basic PC packet. As shown in Figure 4, this message consists of destination and source addresses, a byte count, the data bytes, and a checksum. Notice how the LSB of the addresses must be a 0. For PC, this bit functions as a R/W direction flag, allowing both masters

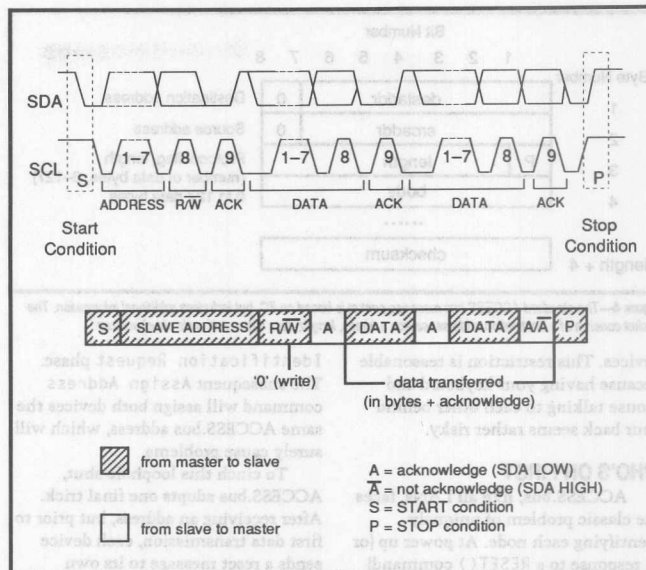


Figure 3—In PC, serial data (SDA) is sampled when the serial clock (SCL) is high. The basic packet of information consists of a destination address, a R/W flag, data, and start/stop framing. A simple ACK/NAK status is sent back by the destination device.

and slaves to function as transmitters and receivers. ACCESS.bus is more restrictive: only a master is allowed to transmit and only a slave may receive. The two-way communication is possible with ACCESS.bus because each device can be a master (when transmitting) or a slave (when receiving) at any given moment.

For example, the host computer is a master when transmitting to a device and a slave when receiving data from that device. So, all messages on the ACCESS.bus are writes from masters to slaves, which is why the R/W flag is fixed at 0 (write).

Packed with the 7-bit data length specifier (i.e., message length = 0 to 127 bytes) the P (Protocol) bit specifies whether the message is a data or status/control transfer. For the latter, ACCESS.bus defines eight messages (four each for computers and devices) as shown in Figure 5. Of these, the most interesting are the commands that pass "capabilities" information from devices to computers.

The capabilities scheme is part of an effort by ACCESS.bus to introduce

a measure of device and software independence. ACCESS.bus groups devices into three generic classes: Keyboard, Locator (e.g., mouse), and Text (e.g., bar-code reader).

The capabilities for a typical mouse might be defined as follows:

```
PROT(LOCATOR)
TYPE(MOUSE)
BUTTONS(1(L)2(R)3(M))
DIM(2) REL RES(200 INCH)
RANGE (-127 127)
DO(DNAME(X))
DI(DNAME(Y))
```

This information describes a device of type "mouse," which uses the generic "locator" device protocol. The 2-D mouse (the dimensions are named X and Y) outputs relative movement between -127 and +127, with a resolution of 200 counts per inch, and has three buttons named L, R, and M.

ACCESS.bus makes a final step back from PC's LAN-like pretensions by allowing transfers only between computers and devices and not among

The Ultimate Desk ACCESSory?

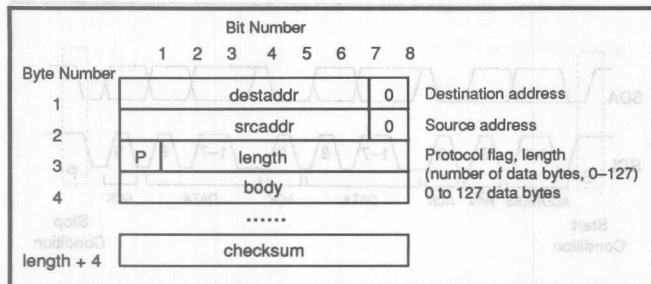


Figure 4—The standard ACCESS.bus message packet is based on PC, but includes additional information. The packet consists of a destination address, source address, length, data bytes, and a simple checksum.

devices. This restriction is reasonable because having your keyboard and mouse talking to each other behind your back seems rather risky.

WHO'S ON FIRST

ACCESS.bus, like all LANs, faces the classic problem of uniquely identifying each node. At power up (or in response to a RESET() command) each ACCESS.bus device reverts to the default address. Next, the computer sends an Identification Request command to the default address (therefore, to all devices on the bus).

At this point, every device will attempt to reply with their 32-bit ID. The ID can be a unique serial number embedded in each device's ROM. However, because this practice adds cost, the protocol also allows devices to generate their own ID, typically via a counter cleared at reset and incremented by the device's internal clock. The result is two of the same devices will usually come up with a different ID thanks to a slight difference in circuit timing.

That all the devices are trying to respond at once is resolved by the previously mentioned multimaster arbitration mechanism. As each ID message gets through, the computer sends an Assign Address command based on the ID. Once the device receives this command, it will assign itself the address specified. Once a device knows its address, it can commence sending and receiving data.

You probably have noticed this procedure has a small, but potentially fatal, loophole: the rare case that two devices report the same ID in the

Identification Request phase. The subsequent Assign Address command will assign both devices the same ACCESS.bus address, which will surely cause problems.

To cinch this loophole shut, ACCESS.bus adopts one final trick. After receiving an address, but prior to first data transmission, each device sends a reset message to its own address. The device sending the message is not itself reset, but any other devices at the same address are. Those devices that are reset will

reenter initialization phases in order to receive new addresses.

TIMING IS EVERYTHING

The proponents of ACCESS.bus are careful to keep reminding us that it is mainly designed for low-speed and low-frequency (i.e., human) input devices. There is a danger of users and suppliers of other I/O devices boarding the bus without a ticket.

Witness the case with the PC's printer port that has been hooked to just about every kind of I/O device including hard disks. The problem is that hooking high-speed block I/O devices could result in a compromised response. Devices like keyboards or mice may not generate a lot of data, but users won't be happy if they don't perceive these devices' responses as instantaneous.

To this end, the specification imposes a number of limits on the amount of traffic or delays any device can impose. For instance, a so-called noninteractive device like a laser printer can only occupy the bus for 5 ms at a time, which limits the maxi-

Computer-to-Device Messages		Purpose
Reset()		Force device to power-up state and default I ² C address.
Identification Request()		Ask device for its "identification string."
Assign Address (ID string, new addr)		Tell device with matching "identification string" to change its address to "new address."
Capabilities Request (offset)		Ask device to send the fragment of its capabilities information that starts at "offset."
Device-to-Computer Messages		
Attention(status)		Inform computer that a device has finished its power-up/reset test and needs to be configured; "status" shall be the test result.
Identification Reply(ID string)		Reply to Identification Request with device's unique "identification string."
Capabilities Reply(offset, data frag)		Reply to Capabilities Request with "data fragment," a fragment of the device's capabilities string; the computer uses "offset" to reassemble the fragments.
Interface Error()		Invalid checksum or premature end of message detected.

Figure 5—An ACCESS.bus message may be either data or status/control. Eight status/control messages are defined, four in each direction.

The Ultimate Desk ACCESSory?

imum data block size to 50 bytes or so (even though the protocol allows up to 127 bytes of data in a message). Furthermore, the device must delay for at least 12 ms after transferring a message before starting another transfer. Finally, abuse of the clock synchronization scheme is prohibited; a device may hold SCL low only for a maximum of 2 ms.

Assuming noninteractive devices obey the rules, interactive devices (i.e., mouse, keyboard, etc.) have plenty of bus available, enough to guarantee 60 Hz (16.6 ms) response. This amount of time is essentially the CRT frame rate, so screen response is fast and smooth.

YAWN?

Is ACCESS.bus a YAWN (Yet Another Wiring and Networking scheme)? The backers of ACCESS.bus are to be commended for avoiding NIH pretensions. PC is quite suitable for the task and clearly has a good laundry list of technical features.

The technical stuff is nice, but it shouldn't be made the focus of too much attention. The fact is, the main strength of ACCESS.bus is that it is a single, open standard offering a solution to PC cable chaos. Energy spent arguing the bits and bytes will only detract from the true battle: overcoming the elephantlike inertia that characterizes the PC market.

The ACCESS.bus proponents have lined up quite a list of suppliers that comprise the infrastructure—cables, connectors, keyboards, mice, and chips—that must underlie any attempt to overcome the powers that be.

Likely, the next step will be the development of PCs that simultaneously support the old interfaces and ACCESS.bus, first with add-in cards and later on the PC motherboards themselves. From there, ACCESS.bus-only systems are just a short hop away.

I hope this move happens soon. When it comes to adding a port or swapping a cable, these old bones are getting pretty tired of "assuming the position." PC owners, rise up off your knees! Here's a chance to go one up on the Mac. ☐

Tom Cantrell has been in Silicon Valley for more than ten years working on chip, board, and systems design and marketing. He can be reached at (510) 657-0264 or by fax at (510) 657-5441.

CONTACT

Philips/Signetics Company
811 East Arques Ave.
Sunnyvale, CA 94088-3409
(800) 227-1817

For an ACCESS.bus developers kit, contact Sharon Baker at (408) 991-3518.



Computer Access Technology's ACCESS.bus standard is being provided for a number of PC and Macintosh systems. The standard is being provided for a number of PC and Macintosh systems.

The board was the same design as the one used in the ACCESS.bus kit. It was designed to be a single board that could be used in a number of different ways. It was designed to be a single board that could be used in a number of different ways. It was designed to be a single board that could be used in a number of different ways.

ACCESS.bus is designed to handle a wide range of devices. It is designed to handle a wide range of devices. It is designed to handle a wide range of devices. It is designed to handle a wide range of devices.

©1992 The Computer Applications Journal.
Reprinted by permission.

ACCESS.bus hardware released for industrial and commercial environments

A Reprint from
**COMPUTER
DESIGN**

COMPUTERS & SUBSYSTEMS

ACCESS.bus hardware released for industrial and commercial environments

Introduced over a year ago, ACCESS.bus has been relatively slow in gathering momentum. The basic idea was to bring some order to the interconnection of a broad range of accessory devices, such as keyboards, locators and bar-code readers.

Digital Equipment Corp and a handful of other OEMs have been im-

time control applications. Initially it was intended to handle up to 14 different devices on a single serial cable, which could be as long as 8 meters. The early specification, however, left some leeway for implementation options, and CATC's approach handles up to 125 devices, while extending the 25 ft to 250 ft

ware drivers. A Windows 3.1 version of the manager program is optional.

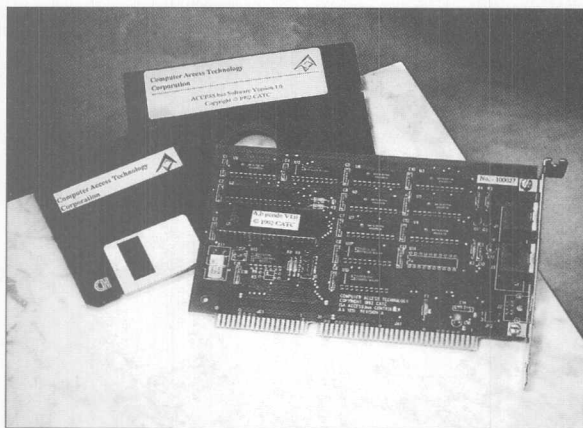
Commercial and industrial

The primary offering of CATC is a standard 16-bit AT/ISA half-board (4.2 x 6.5-in.) design that fits in a single slot and provides two ACCESS.bus connectors. An 8k x 8-bit SRAM buffer memory is included on the board. The unit controls a standard ACCESS.bus network, which lets you add peripherals in several ways.

CATC has also recently added to its product line a compact PC/104 controller to handle industrial applications. The unit has a smaller form factor (3.6 x 3.8-in.) but is functionally the same as the AT/ISA version. Says CATC president Dan Wilnai, "The PC/104 ACCESS.bus module exploits two new open standards ideally suited to embedded control applications. The ultra-compact, stackable module architecture of the PC/104 standard makes it easy to design the full capability of a PC into all kinds of equipment and instrumentation, while the ACCESS.bus serial bus communication standard based on the I²C physical layer protocol lets the designer connect multiple sensors or actuator devices to a single I/O port."

At the electrical level, ACCESS.bus functions as Philips initially defined its I²C setup. The host and devices are connected to both the data and clock lines in a "wired-AND" logical configuration. The wired-AND is implemented by connecting the data and clock output stages of each bus node to the lines through open-collector or open-drain transistors. These devices are included on existing I²C components. The wired-AND configuration lets any of the bus nodes force either line low. When there's no output from any bus node, the lines are held high with pull-up current sources in the host. All devices sense the level on both the clock and data lines.

Because of its relatively straightforward implementation and arbitration based on the same wired-AND configuration, the bus is basically immune to disruptions from the live insertion of additional peripherals.



Computer Access Technology's ACCESS.bus standard AT board provides for the connection of up to 125 peripheral devices to a single communications port in a PC/AT computer. An ACCESS.bus development support kit is also available. The controller board is co-marketed with Philips.

plementing ACCESS.bus, but only recently has it started to catch on. Earlier this year, for example, an ACCESS.bus industry group was formed by 22 manufacturers. In addition, a company has been founded to exploit the benefits of ACCESS.bus. Computer Access Technology Corp (CATC—Sunnyvale, CA) has begun introducing products that let up to 125 peripheral devices connect to a single communications port in a PC/AT system.

ACCESS.bus is designed to handle relatively slow computer input from accessory devices such as keyboards, mice, bar-code readers, magnetic-card readers, modems, and some signal transducers for real-

time control applications.

The board uses the basic Signetics I²C configuration and interface, with an integral Signetics 8XC654 microcontroller; it can handle data rates of up to 80 kbits/s (100 kbits/s minus overhead). In addition, the ACCESS.bus cable carries a +5-V supply voltage for powering each device. The cable carries up to 1 Amp.

CATC's board is supported by an extensive software package, including on-board microcode to control physical ACCESS.bus devices and an ACCESS.bus manager program that runs as a TSR under the PC/AT DOS operating system, routing control and applications messages between the physical devices and their soft-

ACCESS.bus hardware released for industrial and commercial environments

COMPUTERS & SUBSYSTEMS

While this is of some value in a commercial or desktop application, it's very important in industrial applications, where it could greatly simplify servicing, monitoring, updating, and downloading information. To avoid rebooting a system in such situations could result in saving appreciable downtime.

CATC offers a development support package that contains a controller board, an ACCESS.bus mouse, expansion box, cables, and an 87C751 microcontroller. The kit also includes a comprehensive software package and user's manual. The software package consists of on-board microcode and the ACCESS.bus manager, as well as a sophisticated ACCESS.bus monitor and control program. Source code for generic device driver interfaces to the PC and for ACCESS.bus device software modules is also included.

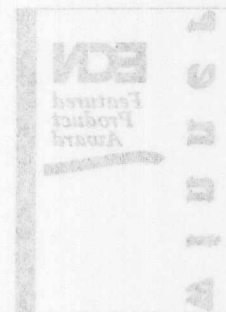
CATC claims that the low price of the controller boards—\$86 for the AT version in lots of 1,000—will go a long way to assure the rapid acceleration of ACCESS.bus technology. Available now, the development support package sells for \$1,500 and the PC/104 modules for \$245.

— Warren Andrews

ACCESS.bus at a glance

- ACCESS.bus controller board.
- Up to 125 peripherals
- Operates at up to 25 ft (250 ft with extender)
- Connects multiple interactive io devices
- Uses low-cost [2]C microcontroller technology
- Compatible with open industry standard
- Connects keyboards, mice, trackballs, digitizers, scanners
- Available as AT/ISA or PC/104
- Development kit available

Computer Access Technology
949 Hillsboro Ave
Sunnyvale, CA 94087
(408) 732-8910



The first ACCESS.bus controller board allowing the connection of up to 125 peripheral devices to a single communications port in a PC/AT computer is available from Computer Access Technology Corporation (CATC).

ACCESS.bus is an open industry standard that provides a simple and uniform way to connect multiple interactive IO devices like keyboards, mice, trackballs, digitizers, scanners, and numerous others to a single computer port.

ACCESS.bus features a data rate of 100,000 bits per second with hardware arbitration and dynamic reconfiguration. It is based on off-the-shelf, low-cost IC microcontroller technology.

ACCESS.bus controller board connects 125 peripherals to a single PC/AT comm port

ECN ELECTRONIC COMPONENT NEWS



The first ACCESS.bus controller board allowing the connection of up to 125 peripheral devices to a single communications port in a PC/AT computer is available from Computer Access Technology Corporation (CATC).

ACCESS.bus is an open industry standard that provides a simple and uniform way to connect multiple interactive I/O devices like keyboards, mice, trackballs, digitizers, scanners, and numerous others to a single computer port.

ACCESS.bus features a data rate of 100,000 bits per second with hardware arbitration and dynamic reconfiguration. It is based on off-the-shelf, low-cost I²C microcontroller technology.

The CATC A.b-125I PC/AT controller board serves as an ACCESS.bus master providing connectivity for desktop as well as instrumentation and control applications. The board, based on the Philips 8xC654 microcontroller with I²C interface, is supported by an extensive software package including on-board microcode to control the operation of physical ACCESS.bus devices, and an ACCESS.bus manager program that runs as a TSR under the PC/AT DOS operating system and routes control and applications messages between the physical devices and their software drivers. A Windows 3.1 version of the manager program can be ordered as an option.

The A.b-125I controller is a highly integrated half-board (4.2" x 6.5") design that fits a single 16-bit AT/ISA slot and provides two physical ACCESS.bus connectors. An 8K x 8-bit SRAM buffer memory is included on the board.

The unit controls a standard ACCESS.bus network supporting up to 125 devices at distances to 25' or 250' with an optional external ACCESS.bus buffer.

To make the development process easier, CATC offers an ACCESS.bus development support package. The A.b-DEV-KIT contains the A.b-125I controller board, an ACCESS.bus mouse, expansion box and cables, and an 87C751 microcontroller. The development kit also includes a

comprehensive software package and a user's manual. The software package consists of the A.b-125I on-board microcode and ACCESS.bus manager, and a sophisticated ACCESS.bus monitor and control program. Source code for generic device driver interfaces on the PC and for ACCESS.bus device software modules is also included.

The monitor can significantly reduce the time and effort spent on software development.

The program lets the user review the behavior of each ACCESS.bus device and connect virtual devices to software drivers. The monitor displays all devices on the bus with their capabilities, identification strings and status. It allows the user to send ACCESS.bus commands manually to any device on the bus and to the A.b-125I controller board. All ACCESS.bus events are recorded in a log file.

CATC. The A.b-125I controller board, the A.b-DEV-KIT ACCESS.bus development support package, and ACCESS.bus accessories are all available from CATC. Controller boards are priced at \$86 each in quantities of 1000. The development support package is \$1500. Contact: Computer Access Technology Corporation, 3375 Scott Blvd. #410 Santa Clara, CA 95054 800-909-CATC (2282) or 408-727-6600 Fax 408-727-6622

I/O Standard Gains Multivendor Support

Author: Ann Miller

OPEN SYSTEMS TODAY

A CMP Publications, Inc. Publication



Grass-roots support is growing for a connectivity standard that could lead to broad compatibility of I/O devices across different computer platforms.

Computer makers have been hesitant to adopt the Access.bus technology, however. But it got a boost recently when Sun Microsystems and Microsoft joined the Access.bus Industry Group (ABIG), a consortium founded last summer to promote the standard. Nevertheless, even committed supporters admit that the standard won't remain viable unless it is adopted by other top systems vendors.

Developed by Digital Equipment Corp. three years ago, the Access.bus method of connecting I/O devices addresses the problem of incompatibility of peripherals across computer system, backers said.

Access.bus employs a standard connector—a plug resembling a wide telephone jack—and a standard firmware specification for I/P communications. Users can "hot plug" an Access.bus device without having to worry whether a communications port is available or whether the right device driver is installed, said Dan Wilnai, president of Computer Access Technology, a Sunnyvale company that makes Access.bus products.

The technology provides cross-system compatibility to low-speed devices, such as modems, mice, printers and scanners, Wilnai said.

Access.bus has caught the attention of peripherals makers, which make up the bulk of the group's membership. They have had to develop different versions of the same device for each computer system.

"It's a real problem, and if Access.bus is accepted by the industry and becomes a standard, it will really benefit us dramatically," said Don Bynum, general manager of Itac

Systems, a Dallas-based maker of industrial trackballs.

Itac has added an Access.bus-compatible trackball to its product line. The company also is a member of the ABIG.

The Access.bus group's membership is composed of about 40 members. The group has no membership fees and provides Access.bus specifications free of charge to any vendor that wants them, said Wilnai, who is ABIG's chairman. At a meeting in January, members decided to seek an IEEE endorsement for the standard, Wilnai said.

The technology is especially promising for peripherals companies that make highly specialized products, such as input devices for the disabled, said Andy MacRae, market segment manager for storage and I/O devices at Sun. Such companies are typically small and often can't afford to produce different versions of their products for different systems, MacRae said.

Users who need specialized devices also could take the devices with them to different sites and plug them into computers they need to use, MacRae said.

Access.bus also offers the ability to connect multiple low-speed devices to a single communications port, Wilnai said. Up to 125 devices can be daisy-chained to a single port on an ISA-based PC using a Computer Access Technology add-in board, Wilnai said. Other experts said Access.bus more commonly supports up to 14 devices.

MULTIPLE DEVICES

The ability to use multiple devices is a benefit to users in many application areas, Wilnai said. For example, he said, users in classrooms could connect multiple keyboards to a single PC. In industrial settings, several employees could use scanners to input data on a single workstation.

Access.bus improves overall performance because it handles interrupts at one per message rather than one per byte of data, Wilnai said. Most serial and parallel devices today must interrupt the CPU for each byte, he said.

Despite apparent benefits to users and peripherals companies—and despite interest

from top industry players—computer vendors have been slow to embrace the technology.

A main reason, observers said, is that vendors don't want to produce Access.bus products before they are sure the standard is viable.

At a time when companies are being conservative in R&D investment, most don't want to risk adopting a technology that might not be widely accepted, said Paul Nelson, senior engineering manager for input devices at DEC and an inventor of the Access.bus.

"They're all just kind of sitting there saying, 'Should we, or shouldn't we?'" Nelson said. "Nobody wants to be the first."

Even DEC, which created the Access.bus using the Philips I²C serial bus technology, has been somewhat noncommittal.

Groups within DEC have been studying and developing and promoting the technology for more than five years, but the company has been reluctant to put it into commercial products, Nelson said.

DEC did implement the Access.bus in its DECstation 5000 family of Unix personal workstations released in December 1991. However, DEC is wary of adopting technologies that could end up being proprietary, and the company has put a hold on further incorporating Access.bus in its systems, Nelson said.

CAUTIOUS APPROACH

Sun also is approaching Access.bus with caution, MacRae said. Though the company has not committed to implementing the technology in its own computers, he said, it is working to pump up user support by cooperating with third-party vendors.

Sun will host the Access.bus group's next quarterly meeting in April. The company also is working with Computer Access Technology on a Sun version of its add-in board, MacRae said.

Wilai said a key strategy for the group is to get PC vendors to incorporate the Access.bus into their systems. Although several vendors have displayed an interest, he said, none has made a commitment to the technology.

Special Report: ACCESS.bus Specs And Products

EE Product News™

An INTERTEC Publication

New Products For Prototype Design

The development of ACCESS.bus, a communications protocol for connecting multiple, low-speed I/O devices to a single computer port, and the formation last June of the ACCESS.bus Industry Group (ABIG) is spawning creation of a rapidly growing number of hardware and software products based on the ACCESS.bus' connectivity specs (see chart below). ACCESS.bus was jointly developed by Digital Equipment Corp. and Philips Semiconductors and is now owned and supported by ABIG, who is promoting the new bus as an industry standard. To date, up to 125 peripheral devices, including keyboards, mice, scanners, digitizers, and bar code readers have been made to operate independently on a single computer port.

At the hardware level, ACCESS.bus uses the I²C (Inter-Integrated Circuit) serial bus developed by Philips several years ago to simplify automotive electronics and other distributed control systems. This serial bus is designed to carry 1 bit of information at a time on a single data line. Today, a host of low-cost I²C components are readily available to handle the logical complications associated with bit-level handshaking.

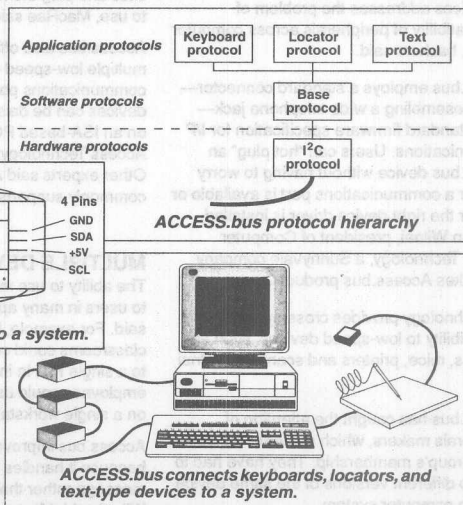
The physical medium for ACCESS.bus is a shielded cable with four wires for handling serial data (SDA), serial clock (SCL), power (5V), and ground (GND). The SDA and SCL lines work together to define information carried on the bus, and the host computer drives the 5V power line with a minimum of 50 mA to supply peripheral devices (peripherals can also be externally powered). A typical ACCESS.bus device has two connectors, permitting two or more peripherals to be daisy-chained together on the bus (handheld devices can have a captive cable joined to the bus trunk with a T-connector).

I²C technology supports clock rates up to 100 kHz and the maximum ACCESS.bus data transfer rate is approximately 80 kbits/s.

The ACCESS.bus communications protocol has three layers: I²C, Base and Applications. The I²C Protocol defines a symmetric, multi-master bus on which arbitration among contending masters is effected without losing data. I²C provides cooperative synchronization of the

serial clock for exchange of data between bus partners with different maximum clock rates, defining a bus transaction scheme with addressing, framing of bits into bytes, and byte acknowledgement by the receiver.

Base Protocol establishes an asymmetrical interconnect between host computer and peripheral devices. The host is the ACCESS.bus manager, and data communication is always between host and peripheral, never between two peripherals. While the I²C Protocol establishes mastership between the sender or receiver of a bus transaction, Base Protocol defines the format of an ACCESS.bus message envelope, which is an I²C bus transaction with



additional semantics, including checksum reliability control. Base Protocol also defines a set of seven control and status message types used in the configuration process.

The high-level Application Protocol defines message semantics specific to particular functional types of devices. To date, Application Protocol have been established for keyboards, locators and text devices and is intended to define the simplest set of functions from common, industry-standard interfaces. Further, device-specific Application Protocol models will be defined by the ACCESS.bus

Special Report: ACCESS.bus Specs And Products

Industry Group; and, of course, any vendor can implement a special device protocol within the general message envelope defined by the Base Protocol.

Electrically, host and peripheral devices are connected to serial data (SDA) and serial clock (SCL) lines in a wired-AND logic configuration, which can be implemented by connecting data and clock output stages of each bus partner to the SDA and SCL lines, respectively, through open-collector or open-drain transistors. Standard I²C components include these output stages on-chip. Significance of the wired-AND logic is that any attached bus partner can force either of these lines to LOW (GND); and when there is no output from any bus partner, lines are held HIGH by pull-up current sources in the host. Every bus partner can sense the level on both of these lines.

ACCESS.bus can be adapted to any platform and presently requires use of a controller board in the computer, but within 12 months, computer motherboards containing the necessary ACCESS.bus circuitry are expected to begin to appear. Software drivers are also available for DOS and Windows.

For additional information on ACCESS.bus v2.0 specs and on membership to ABIG, contact:
ACCESS.bus Industry Group, 415-112 N. Mary Ave., Sunnyvale, CA 94086, (408) 991-3517, FAX (408) 991-3773.

length of the message body.

ACCESS.bus: A New Peripheral Bus

Author: Michael Burton

Midnight™ Engineering

Would you like to be able to plug a keyboard, two mice, a trackball, a modem and a printer into a single port on your PC? And then, while the computer is still powered up, unplug one of the mice and plug in a bar code reader? You can do all of this and more with ACCESS.bus.

Digital Equipment Corporation and Philips have joined forces to propose this new open desktop connectivity standard. The ACCESS.bus Industry Group (ABIG) has been formed to regulate and promote the new bus. ABIG members include DEC, Honeywell, Logitech, Philips and Sun Microsystems, to name just a few.

There are many advantages to the ACCESS.bus standard. It is low cost, dynamically reconfigurable, relatively inexpensive and the interface is uniform for all devices. ACCESS.bus is also an open standard, unlike Apple Computer's comparable Apple Data Bus (ADB).

Bus Description

ACCESS.bus allows multiple peripheral devices to be simultaneously supported on a single computer port in a daisy chain fashion, somewhat like a SCSI daisy chain. These peripherals may operate simultaneously at transfer rates of up to 125,000 baud, with a maximum data throughput rate of approximately 80,000 baud. This is ideal for low speed peripherals such as keyboards, modems, trackballs and mice.

A maximum cable length of 8 meters is permitted. A four pin, shielded rectangular connector is used on ACCESS.bus cables. The maximum amount of power available is 1 amp at 5 volts. The bus can support up to 125 peripheral devices, but the normal practical limit is 14. By way of comparison, the Apple Data Bus supports 5 meters of cable and 3 devices with a data rate of 10,000 baud.

ACCESS.bus is a layered protocol. There are three layers: the hardware protocol layer, the Base protocol layer and the Application

protocol layer. Using a postal analogy, the hardware protocol layer is the mail carrier, the Base protocol layer is the envelope and the Application protocol layer is the contents of the envelope.

The hardware protocol layer is based on the I²C (Inter-Integrated Circuit) serial protocol, which is directly supported by the Philips 8051 family of microprocessors (80CL410, 80C552, 80C652, 80C528, 87C654 and 87C751). This protocol defines a scheme for performing bus transactions, including message addressing, the framing of bits into bytes and the acknowledgment of each byte by the receiver.

The Base protocol layer is a software protocol that is common to all ACCESS.bus devices and that builds on the hardware protocol layer to establish the connection between the computer and a number of peripheral devices. The Base protocol layer specifies device power-up, identification, addressing and the message envelope for device-specific data and control information.

The Application protocol layer is a software protocol that differentiates message contents for specific kinds and classes of devices.

Hardware Protocol Layer

The hardware I²C protocol layer is a 2-wire (clock and data) serial protocol that allows wire-AND connection of peripheral devices to the clock and data lines. Any device may be either a master (controlling the transaction and generating the clock) or a slave for any given bus transaction. Several masters can contend for the bus and an arbitration scheme resolves bus mastership without data loss or retransmission. The clock rates for various peripherals may vary widely, since the bus has a cooperative serial clock synchronization scheme.

Base Protocol Layer

The Base protocol layer contains definitions for a number of control and status messages that are common to all ACCESS.bus peripherals. The messages are used for the

configuration process, where peripherals are recognized, assigned unique address identifiers and are then connected with appropriate device drivers to enable an application program to talk to them. A message has five parts to it:

1. The first byte is the address of the destination or receiver.
2. The second byte is the address of the source or transmitter.
3. The third byte specifies whether the body of the message is control or data, if there are any sub-devices (0 to 3) and the length of the message body.
4. This part is the message body, which can be from 0 to 127 bytes in length.
5. This last byte is the checksum, a bit-wise exclusive-or of all the preceding bytes in the message.

There are eight base messages, shown below.

Computer to Device Messages

1. Reset—Force the device to its power-up state and to its default I²C address.
2. IdRequest—Ask the device for its identification string.
3. AssignAddress—Tell the device with a matching identification string to change its address to a new address.
4. CapRequest—Ask the device to send its capabilities information.

Device to Computer Messages

1. Attention—Inform the computer that the device has finished its power-up/reset tests and that it needs to be configured.
2. IdReply—Reply to an IdRequest with the device's unique identification string.
3. CapReply—Reply to a CapRequest with a fragment of the device's capabilities string.
4. IfError—Invalid checksum or premature end of message detected.

ACCESS.bus: A New Peripheral Bus

When a peripheral device powers up or resets, its initial device address is always 6Eh. The Base messages are used to reset this device address to a unique address between 02h and 7Eh (125 assignable addresses).

Application Protocol Layer

Application protocol layer messages are specific to the peripheral device and the message layouts are different for each device type, as well as for each device sub-type. This means that the device drivers are different at this level, but since the hardware and Base protocols are device-independent, much of the firmware support code can be shared by different devices. Even at the Applications protocol level, a common message structure is used for similar device types, so that a common approach can be used in writing ACCESS.bus device drivers. To date, peripheral devices for ACCESS.bus have been classed into three broad categories:

Keyboards—May have as many as 255 keys. Special function keys and annunciators are supported.

Locator devices—Includes pointing devices

such as mice, trackballs, graphic tablets, etc. Provides for devices with up to 15 degrees of freedom and up to 16 binary keys.

Text devices—Devices that support data streams (e.g., modems, printers, etc.).

What ACCESS.bus Means to You

ACCESS.bus is a coherent, well-defined desktop connectivity standard. It is solidly backed by DEC, Philips and others in the industry and it offers much to both users and peripheral manufacturers. Since it is so new, there are plenty of opportunities for entrepreneurs to implement ACCESS.bus devices with fewer direct challenges from the big guys. Peripheral devices that already use an 8051 microprocessor only need to change their I/O firmware, so the impact of designing for a new bus can be minimized. Watch out, though—Microsoft and others are starting to pay attention to ACCESS.bus, so the window of opportunity is getting smaller.

ACCESS.bus Access

The ACCESS.bus Industry Group will provide free information, including the ACCESS.bus Specification, to anyone who asks for it. It

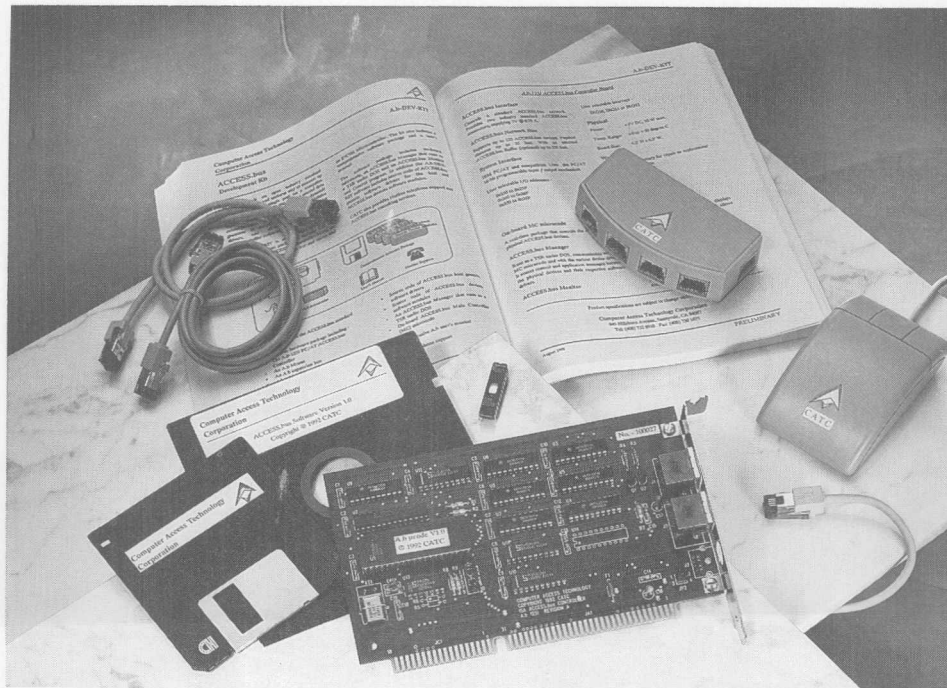
costs nothing to join ABIG (as a company) if you plan to develop an ACCESS.bus peripheral. ABIG's address is:

ACCESS.bus Industry Group
370 Altair Way Suite 215
Sunnyvale, CA 94086
408-991-3517 or FAX: 408-991-3773

At least one company provides and ACCESS.bus Development Kit, at a cost of \$1500. The kit includes a controller board, a Logitech ACCESS.bus mouse, an expansion box, two cables, a Philips 87C751 microprocessor, a comprehensive software package and documentation for everything. Their address is:

Computer Access Technology Corporation
3375 Scott Blvd. #410
Santa Clara, California 95054
800-909-CATC (2282)
408-727-6600
FAX: 727-6622

Michael Burton is a Senior Software Engineer, at Key Tronic Corporation, Spokane, WA.



Section 4

Other Application Notes & Articles

INDEX

AN408	80C451 operation of port 6	4-3
AN417	256k Centronics printer buffer using the 87C451 microcontroller	4-14
AN418	Counter/timer 2 of the 83C552 microcontroller	4-27
AN420	Using up to 5 external interrupts on 80C51 family microcontrollers	4-34
AN424	8051 family warm boot determinations	4-36
AN440	RAM loader program for 80C51 family applications	4-38
AN443	IEEE Micro Mouse using the 87C751 microcontroller	4-49
AN447	Automatic baud rate detection for the 80C51	4-70
AN448	Determining baud rates for 8051 UARTs and other UART issues	4-73
ESG89001	Electro magnetic compatibility and printed circuit board (PCB) constraints	4-76
EIE/AN91001	Workbench EMC evaluation method	4-104
EIE/AN91006	A/D conversion with P83CL410 PCF1252-x	4-121
EIE/AN91009	Driver for 8xC851 E2PROM	4-139
EIE/AN92001	Low RF-emission applications with a P83CE654 microcontroller	4-162
Chips push CAN bus into embedded world		4-174
Add Text Overlay to Any Video Display		4-176

80C451 operation of port 6

AN408

INTRODUCTION

The features of the 80C451 are shared with the 80C51 or are conventional except for the operation of port 6. The flexibility of this port facilitates high-speed parallel data communications. This application note discusses the use of port 6 and is divided into the following sections:

1. Port 6 as a processor bus interface.
2. Using port 6 as a standard pseudo bidirectional I/O port.
3. Implementation of parallel printer ports.

This information applies to all versions of the part: 80C451, 83C451, and the 87C451.

PORT 6 AS A PROCESSOR BUS INTERFACE

Port 6 allows use of the 80C451 as an element on a microprocessor type bus. The host processor could be a general purpose MPU or the data bus of a microcontroller like the 80C451 itself. This feature allows single or multiple 80C451 controllers to be used on a bus as flexible peripheral processing elements. Applications could include keyboard scanners, serial I/O controllers, servo controllers, etc.

OPERATION

On reset, port 6 is programmed correctly for use as a bus interface (see 2). This prevents the interface from disrupting data on the bus of the host processor during power-up. Software initialization of the CSR (Control Status Register) is not required. A dummy read of port 6 may be required to clear the IBF (Input Buffer Full) flag since it could be set by turn on transients on the bus of the host processor. On reset, the CSR of the 83C451 is programmed to allow the following:

1. AFLAG is an input controlling the port select function. If AFLAG is high, the contents of the CSR is output on port 6 when the port is read by the host. If AFLAG is low, then the contents of the output latch is output when port 6 is read by the host.
2. BFLAG is an input controlling the port enable function. In this mode when BFLAG is high, the input latch and the output drivers are disabled and the flags are not affected by the IDS (Input Data Strobe) or ODS (Output Data Strobe) signals. When BFLAG is low, the port is enabled for reading and writing under the control of IDS and ODS pins.

Figure 1 shows one possible example of an 80C451 on a memory bus. This arrangement allows the main processor to query port 6 for flag status without interrupting the 80C451. If the address decoder, shown in Figure 1, enables port 6 on the 80C451 when the address is 8000H or 8001H, and the address line A0 controls the port select feature, then the host processor can read and write to port 6 using address 8000H. Since the port select function is being controlled by the address line A0, the CSR contents can be read by the host processor at address 8001H.

By testing the CSR contents in this way, the host processor can tell if new data has been written to the port 6 output latch since it last read the port or if the 80C451 has read the last byte that the host wrote to the port. Conversely, the 80C451 can poll the flags in its CSR to see if the host processor has written to or read from port 6 since the last time it serviced the port.

If desired, an interrupt source for the 80C451 can be derived easily from the port enable source as shown by the dashed line in Figure 1.

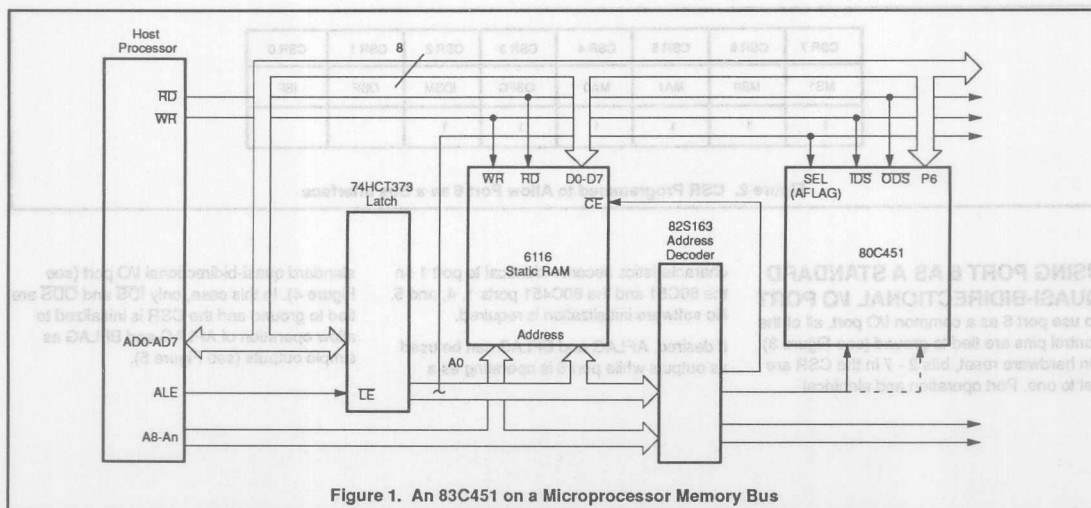


Figure 1. An 83C451 on a Microprocessor Memory Bus

80C451 operation of port 6

AN408

SOFTWARE EXAMPLES

To write to port 6 on the bus shown in Figure 1, the host processor first reads the CSR contents at address 8001H, and tests

the input buffer full flag (CSR bit 0). If the flag is clear, the host writes a byte to address 8000H. This loads the input buffer latch of port 6 and sets the input buffer full flag.

Conversely, the 80C451 polls the IBF flag and reads a byte from port 6 when it finds the flag set. The flag is automatically reset when this internal read occurs.

80C451 ROUTINE TO READ ONE BYTE FROM HOST VIA PORT 6

```
RCVR: JNB CSR.0,RCVR ;TEST IBF FLAG
      MOV A,P6 ;WHEN FLAG IS SET READ BYTE
      RET
```

80C451 ROUTINE TO WRITE ONE BYTE TO THE 80C451 PORT 6

If the host processor is an 80C51, the following routine will write a byte of data to the 80C451. The data involved is passed to the routine through register 1.

```
XMIT: MOV DPTR,8001H ;READ THE CSR
      TEST: MOVX A,@DPTR ;TEST IBF FLAG
           JB ACC.0,TEST ;WRITE DATA TO THE 451
           MOV DPTR,8000H
           MOV A,R1
           MOVX @DPTR,A
           RET
```

80C451 ROUTINE TO WRITE ONE BYTE TO HOST VIA PORT 6

Routines for data transfer in the opposite direction are similar to the above two. The 80C451 version is given below.

```
XMIT: JB CSR.1,XMIT ;TEST OBF FLAG
      MOV P6,A ;WRITE DATA
      RET
```

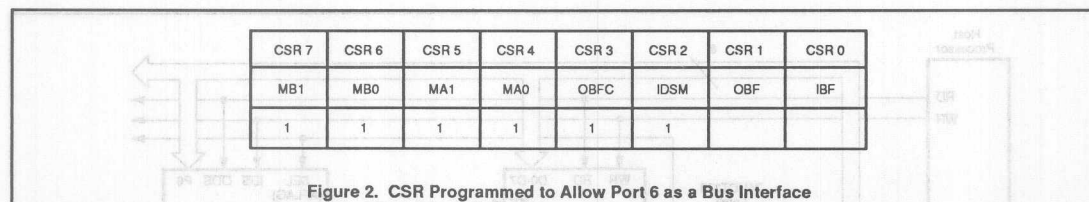


Figure 2. CSR Programmed to Allow Port 6 as a Bus Interface

USING PORT 6 AS A STANDARD QUASI-BIDIRECTIONAL I/O PORT

To use port 6 as a common I/O port, all of the control pins are tied to ground (see Figure 3). On hardware reset, bits 2 - 7 in the CSR are set to one. Port operation and electrical

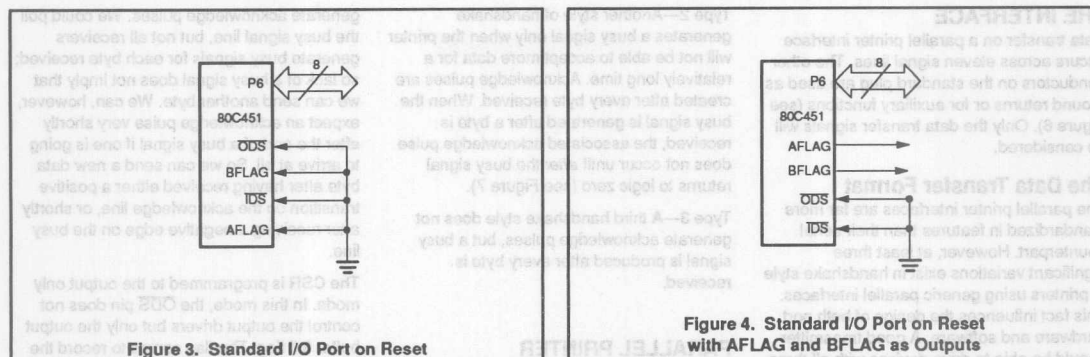
characteristics become identical to port 1 on the 80C51 and the 80C451 ports 1, 4, and 5. No software initialization is required.

If desired, AFLAG and BFLAG can be used as outputs while port 6 is operating as a

standard quasi-bidirectional I/O port (see Figure 4). In this case, only IDSM and ODS are tied to ground and the CSR is initialized to allow operation of AFLAG and BFLAG as simple outputs (see Figure 5).

80C451 operation of port 6

80C451 operation of port 6 AN408



CSR 7	CSR 6	CSR 5	CSR 4	CSR 3	CSR 2	CSR 1	CSR 0
MB1	MB0	MA1	MA0	OBFC	IDS	OB	IBF
1	X	0	X	X	1		

Figure 5. CSR Programmed to Allow AFLAG and BFLAG to Operate as Outputs and Port 6 as a Standard I/O Port

DATA TRANSFER SIGNAL PINS			TYPICAL AUXILIARY PIN FUNCTIONS	
Pin No.	Ground Return Pin No.	Signal	Pin No.	Signal
1	19	STROBE	12	PAPER OUT
2	20	DATA 1	14	AUTO LINE FEED
3	21	DATA 2	16	LOGIC GROUND
4	22	DATA 3	17	CHASSIS GND
5	23	DATA 4	30	GROUND RETURN
6	24	DATA 5	31	RESET PRINTER
7	25	DATA 6	32	ERROR
8	26	DATA 7	33	GROUND RETURN
9	27	DATA 8	36	SLCT IN
10	28	ACKNLG		
11	29	BUSY		

Figure 6. Parallel Printer Interface Pin Functions

IMPLEMENTATION OF PARALLEL PRINTER PORTS USING PORT 6

The 80C451 is an excellent choice for a printer controller. The 80C451 has the facilities to permit all of the intelligent features of a common printer to be handled by a single chip:

1. The features of port 6 allow a parallel printer port to be designed with only line driving and receiving chips required as additional hardware.

2. The onboard UART allows RS232 interfacing with only level shifting chips added.
3. The 8-bit parallel ports 0 to 6 are ample to drive onboard control functions, even when ports are used for external memory access, interrupts, and other functions.
4. The RAM addressing ability of ports 0 and 2 can be used to address up to 64k bytes of a hardware buffer/spooler. AFLAG and BFLAG as simple outputs (see Figure 5).

5. The 64k byte ROM addressing capability allows space for the most sophisticated software.

In addition, either end of a parallel interface can be implemented using port 6, and the interfaces can be interrupt driven or polled in either case.

80C451 operation of port 6

AN408

THE INTERFACE

Data transfer on a parallel printer interface occurs across eleven signal lines. The other conductors on the standard plug are used as ground returns or for auxiliary functions (see Figure 6). Only the data transfer signals will be considered.

The Data Transfer Format

The parallel printer interfaces are far more standardized in features than their serial counterpart. However, at least three significant variations exist in handshake style in printers using generic parallel interfaces. This fact influences the design of both port hardware and software. A good transmitter should be able to drive devices with all three styles of handshakes, and a good receiver should generate the handshake most likely compatible with any transmitter.

The Variations

Type 1—Figure 7 shows a common style of handshake and is the style that will be implemented in the receiver examples. A busy signal and an acknowledge strobe pulse are generated for every byte received.

Type 2—Another style of handshake generates a busy signal only when the printer will not be able to accept more data for a relatively long time. Acknowledge pulses are created after every byte received. When the busy signal is generated after a byte is received, the associated acknowledge pulse does not occur until after the busy signal returns to logic zero (see Figure 7).

Type 3—A third handshake style does not generate acknowledge pulses, but a busy signal is produced after every byte is received.

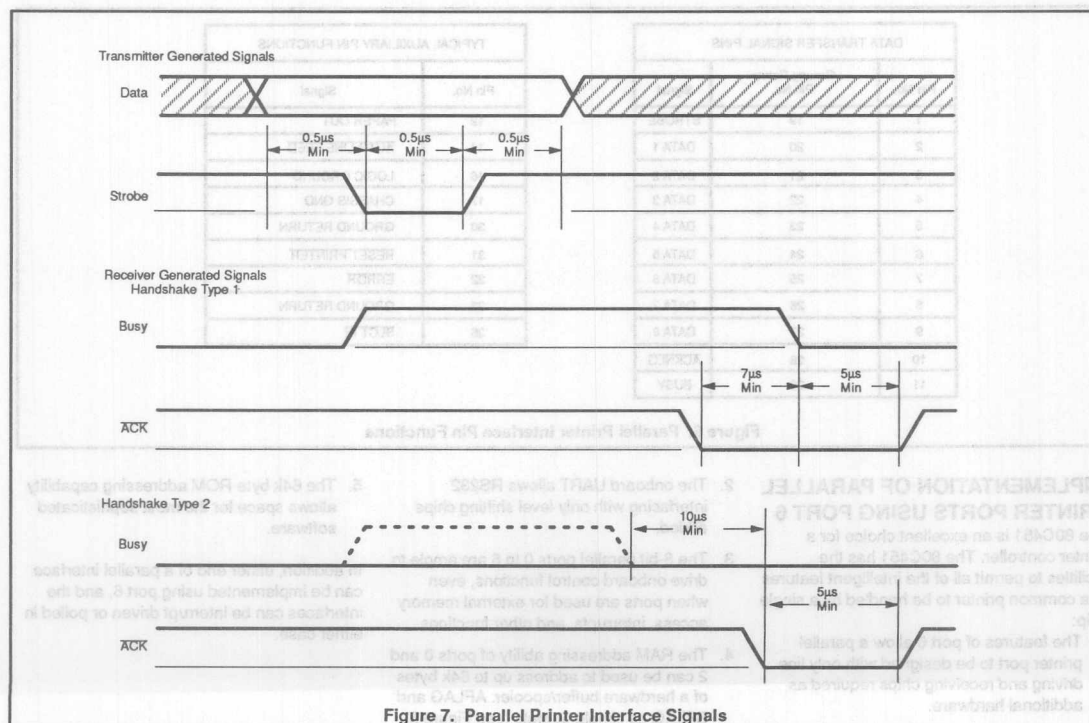
PARALLEL PRINTER INTERFACES USING POLLING

Transmitter Operation

This application illustrates the flexibility of the port 6 logic in solving an applications problem. We need to be able to handle all types of acknowledge signals that might be received by the transmitter. We will use the ODS pin and output buffer full flag logic to record the receipt of the acknowledge pulse (see Figure 8), but not all parallel receivers

generate acknowledge pulses. We could poll the busy signal line, but not all receivers generate busy signals for each byte received; so lack of a busy signal does not imply that we can send another byte. We can, however, expect an acknowledge pulse very shortly after the end of a busy signal if one is going to arrive at all. So we can send a new data byte after having received either a positive transition on the acknowledge line, or shortly after receiving a negative edge on the busy line.

The CSR is programmed to the output only mode. In this mode, the ODS pin does not control the output drivers but only the output buffer full flag. The flag serves to record the positive transition of the acknowledge signal. The input latch is not used, but the IDS pin is used to set the input buffer full flag. This is used to record the negative transition at the end of the busy signal. Dummy reads by the 80C451 of port 6 will be used to clear the flag. In this example, the AFLAG mode is set only to place the port in the output only mode. The AFLAG pin is not actually used (see Figure 10).



80C451 operation of port 6

AN408

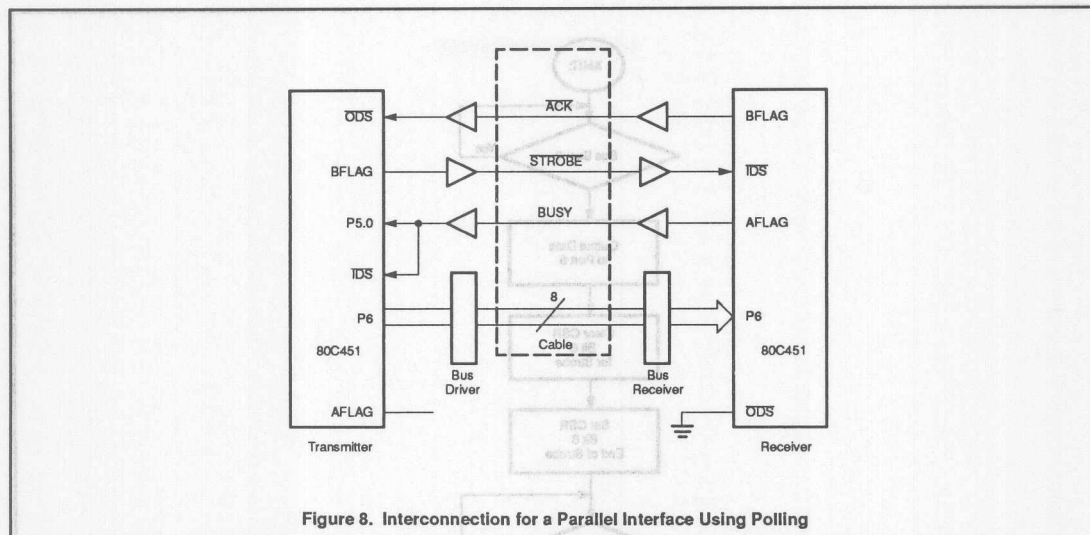


Figure 8. Interconnection for a Parallel Interface Using Polling

CSR 7	CSR 6	CSR 5	CSR 4	CSR 3	CSR 2	CSR 1	CSR 0
MB1	MB0	MA1	MA0	OBFC	IDSM	OBF	IBF
0	1	1	0	0	1		

Figure 9. CSR Programmed for Polled Transmitter Operation

The transmitter's CSR (control status register) is programmed to the following mode (see Figure 9):

1. CSR bit 6 controls the BFLAG output and therefore the strobe line.
2. The OBF (output buffer full) flag controls the AFLAG output.
3. The OBF is cleared on the positive edge of the ODS input.
4. The IBF flag is cleared on the negative edge of the IDS strobe.

NOTE:

With this combination of modes set, port 6 is in the output only mode.

Receiver Operation

In receiver operation, the IDS input is used to latch in the data transmitted on receipt of the strobe pulse. The receiver's CSR is programmed to allow the following (see Figure 11):

1. The input buffer full flag is output through the BFLAG pin and is used as the busy signal to the transmitter.
2. The IBF flag is set and data is latched on the positive edge of IDS.
3. Writing to the CSR bit 4 controls the AFLAG output and therefore the acknowledge line.

80C451 operation of port 6

80C451 operation of port 6

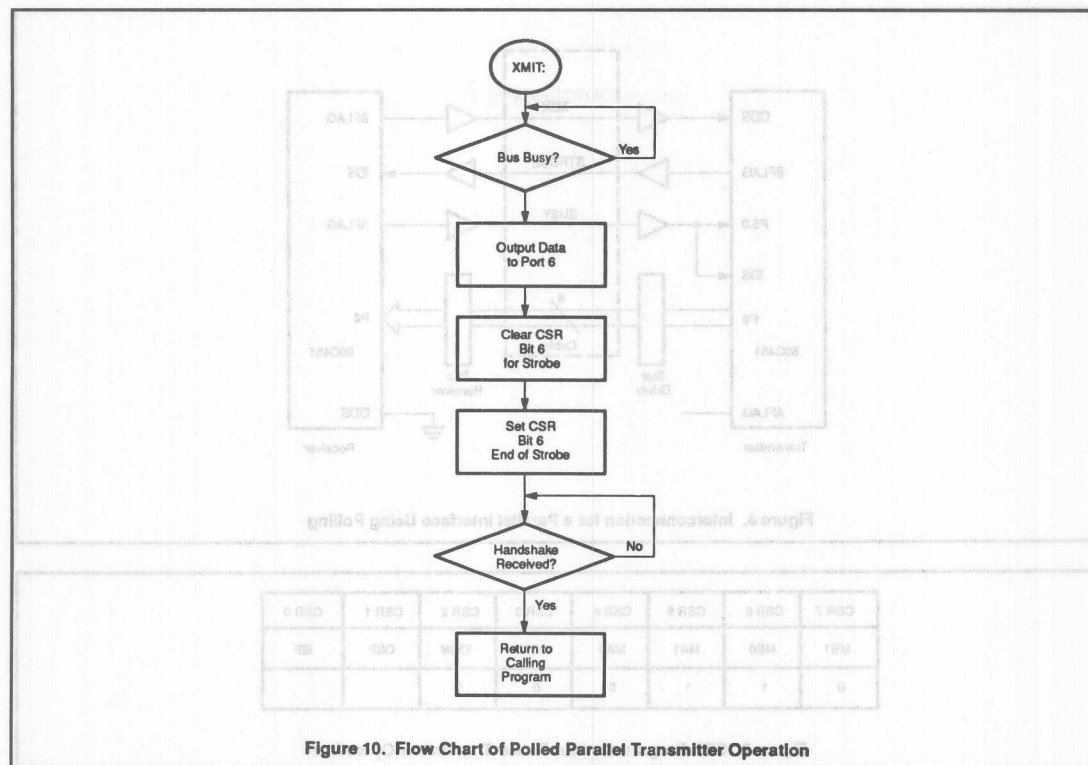


Figure 10. Flow Chart of Polled Parallel Transmitter Operation

CSR 7	CSR 6	CSR 5	CSR 4	CSR 3	CSR 2	CSR 1	CSR 0
MB1	MB0	MA1	MA0	OBFC	IDSM	OB	IBF
1	0	0	1	1	0		

Figure 11. CSR Programmed for Polled Parallel Receiver Operation

80C451 operation of port 6

AN408

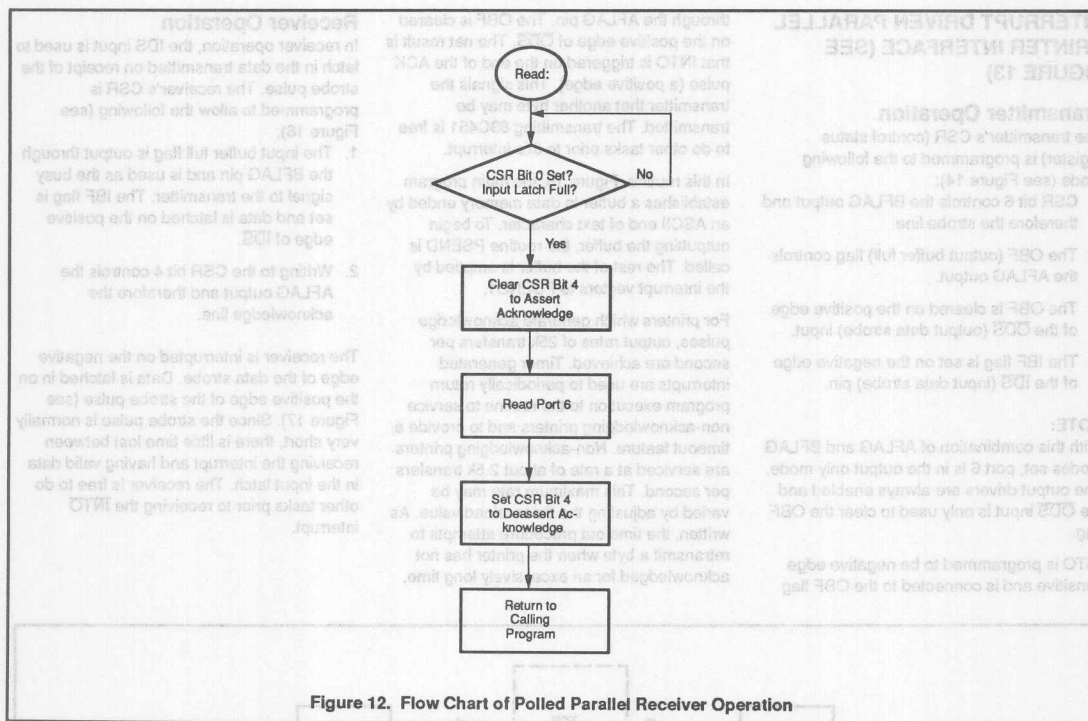


Figure 12. Flow Chart of Polled Parallel Receiver Operation

SOFTWARE EXAMPLES

This polled parallel transmit routine outputs one byte passed to it in the accumulator.

```

P_INIT:  MOV CSR,#064H      ;INITIALIZE PORT 6 OPERATING MODE
P_OUT:   JB P5.0            ;WAIT IF BUSY SIGNAL IS HIGH
        MOV P6,ACC         ;OUTPUT DATA
        MOV R1,P6          ;DUMMY READ TO CLEAR IBF FLAG
        MOV R1,#02H        ;INITIALIZE DELAY COUNTER
        CLR CSR.6          ;START STROBE PULSE
        DJNZ R1,$          ;TIME 6 MICROSECOND STROBE PULSE
        SETB CSR.6         ;END STROBE PULSE
WAIT:    JNB CSR.1,OUT      ;EXIT IF ACKNOWLEDGE RCV'D
        JNB CSR.0,WAIT     ;EXIT IF NEGATIVE BUSY EDGE RCV'D
  
```

This polled parallel receive routine places one byte in the accumulator each time it is called.

```

P_INIT:  MOV CSR,#09CH      ;INITIALIZE PORT 6 OPERATING MODE
        MOV R7,P6          ;DUMMY READ TO CLEAR IBF FLAG
P_IN:    JNB CSR.0          ;INPUT BUFFER LATCH FULL?
        CLR CSR.4          ;BEGIN ACKNOWLEDGE PULSE
        MOV R7,#02H        ;INITIALIZE DELAY COUNTER
        DJNZ R7,$          ;TIME ACKNOWLEDGE PULSE
        MOV A,P6           ;READ BYTE - CLEAR BUSY SIGNAL
        MOV R7,#02H        ;INITIALIZE DELAY COUNTER
        DJNZ R7,$          ;TIME ACKNOWLEDGE PULSE
        SETB CSR.4         ;END ACKNOWLEDGE PULSE
        RET
  
```


80C451 operation of port 6

AN408

**INTERRUPT DRIVEN PARALLEL
PRINTER INTERFACE (SEE
FIGURE 13)**

Transmitter Operation

The transmitter's CSR (control status register) is programmed to the following mode (see Figure 14):

1. CSR bit 6 controls the BFLAG output and therefore the strobe line.
2. The OBF (output buffer full) flag controls the AFLAG output.
3. The OBF is cleared on the positive edge of the $\overline{\text{ODS}}$ (output data strobe) input.
4. The IBF flag is set on the negative edge of the $\overline{\text{IDS}}$ (input data strobe) pin.

NOTE:

With this combination of AFLAG and BFLAG modes set, port 6 is in the output only mode. The output drivers are always enabled and the $\overline{\text{ODS}}$ input is only used to clear the OBF flag.

INTO is programmed to be negative edge sensitive and is connected to the OBF flag

through the AFLAG pin. The OBF is cleared on the positive edge of ODS. The net result is that INTO is triggered on the end of the ACK pulse (a positive edge). This signals the transmitter that another byte may be transmitted. The transmitting 83C451 is free to do other tasks prior to this interrupt.

In this routine, Figure 15, the main program establishes a buffer in data memory ended by an ASCII end of text character. To begin outputting the buffer, the routine PSEND is called. The rest of the buffer is emptied by the interrupt vectors to PSEND1.

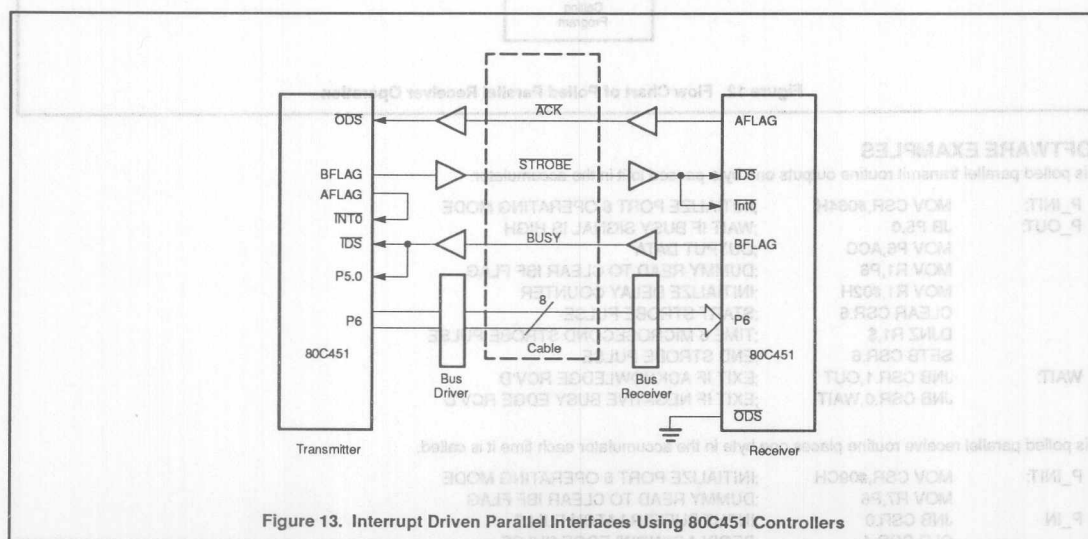
For printers which generate acknowledge pulses, output rates of 25k transfers per second are achieved. Timer generated interrupts are used to periodically return program execution to the routine to service non-acknowledging printers and to provide a timeout feature. Non-acknowledging printers are serviced at a rate of about 2.5k transfers per second. This maximum rate may be varied by adjusting the timer reload value. As written, the time out procedure attempts to retransmit a byte when the printer has not acknowledged for an excessively long time.

Receiver Operation

In receiver operation, the IDS input is used to latch in the data transmitted on receipt of the strobe pulse. The receiver's CSR is programmed to allow the following (see Figure 16):

1. The input buffer full flag is output through the BFLAG pin and is used as the busy signal to the transmitter. The IBF flag is set and data is latched on the positive edge of IDS .
2. Writing to the CSR bit 4 controls the AFLAG output and therefore the acknowledge line.

The receiver is interrupted on the negative edge of the data strobe. Data is latched in on the positive edge of the data pulse (see Figure 17). Since the strobe pulse is normally very short, there is little time lost between receiving the interrupt and having valid data in the input latch. The receiver is free to do other tasks prior to receiving the **INT0** interrupt.



80C451 operation of port 6

AN408

CSR 7	CSR 6	CSR 5	CSR 4	CSR 3	CSR 2	CSR 1	CSR 0
MB1	MB0	MA1	MA0	OBFC	IDSM	OBF	IBF
0	1	1	0	1	1	0	1

Figure 14. CSR Programmed for Use as an Interrupt Driven Parallel Transmitter

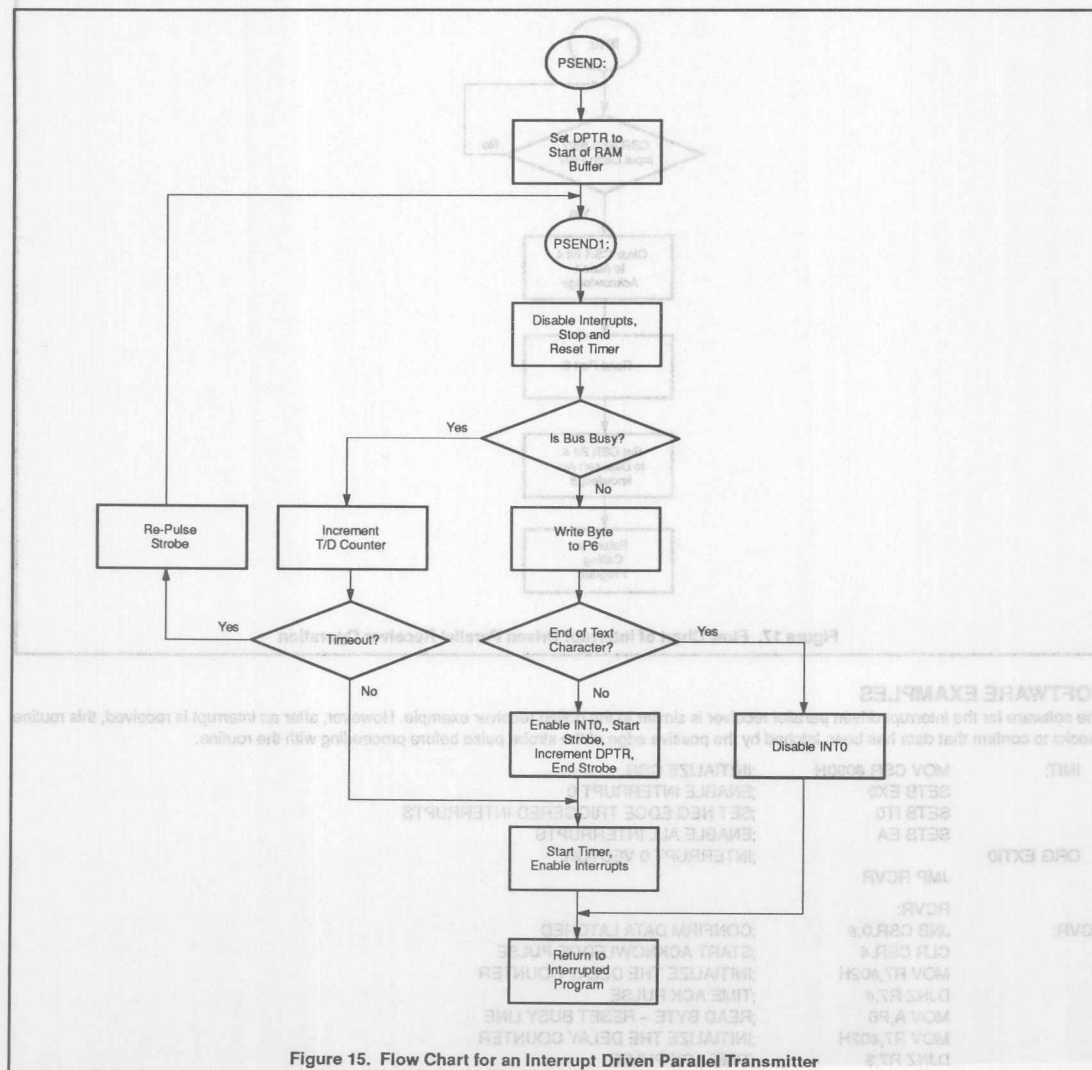


Figure 15. Flow Chart for an Interrupt Driven Parallel Transmitter

80C451 operation of port 6

AN408

CSR 7	CSR 6	CSR 5	CSR 4	CSR 3	CSR 2	CSR 1	CSR 0
MB1	MB0	MA1	MA0	OBFC	IDSM	OBF	IBF
1	0	0	1	1	0		

Figure 16. CSR Programmed for Use as an Interrupt Driven Parallel Receiver

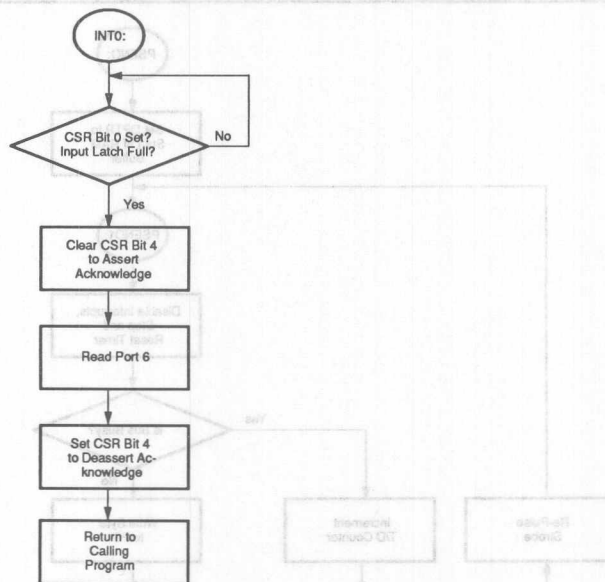


Figure 17. Flow Chart of Interrupt Driven Parallel Receiver Operation

SOFTWARE EXAMPLES

The software for the interrupt driven parallel receiver is similar to the polled receiver example. However, after an interrupt is received, this routine checks to confirm that data has been latched by the positive edge of the strobe pulse before proceeding with the routine.

```

INIT:      MOV CSR,#090H      ;INITIALIZE CSR
           SETB EX0           ;ENABLE INTERRUPT 0
           SETB IT0           ;SET NEG EDGE TRIGGERED INTERRUPTS
           SETB EA            ;ENABLE ALL INTERRUPTS
           ORG EXTIO          ;INTERRUPT 0 VECTOR

RCVR:      JMP RCVR           ;RECEIVE DATA

RCVR:      JNB CSR.0,#        ;CONFIRM DATA LATCHED
           CLR CSR.4          ;START ACKNOWLEDGE PULSE
           MOV R7,#02H        ;INITIALIZE THE DELAY COUNTER
           DJNZ R7,#          ;TIME ACK PULSE
           MOV A,P6           ;READ BYTE - RESET BUSY LINE
           MOV R7,#02H        ;INITIALIZE THE DELAY COUNTER
           DJNZ R7,$          ;TIME ACK PULSE
           SETB CSR.4         ;END ACK PULSE
           RET1
  
```

80C451 operation of port 6

AN408

This is the software for the interrupt driven parallel transmitter example.

; XMIT ROUTINE DRIVEN BY ACK PULSE GENERATED INTERRUPTS, OR TIME GENERATED INTERRUPTS
; FOR NON ACKNOWLEDGING PRINTERS. READS DATA BUFFER IN EXTERNAL RAM STARTING AT 100H
; AND READING UNTIL 04H IS FOUND.

```

ORG RESET
JMP 26H
ORG TIMERO
JMP PSEND1
ORG EXTIO
JMP PSEND1
ORG 26H
    MOV CSR,#064H      ;PORT 6 MODE
    MOV TMOD,#002H     ;CONFIGURE TIMER 0 TO 16 BITS
    SETB T00           ;INT0 IS EDGE TRIGGERED
    SETB EA            ;ENABLE INTERRUPTS
    PSEND: MOV DPTR,#0100H ;SET DPTR TO START OF TEXT
    PSEND1: CLR EA      ;BUFFER
                ;DISABLE INTERRUPTS AND STOP
                ;TIMER
                ;IF ENABLED
    CLR TRO
    CLR ET0
    MOV R7,00H      ;CLEAR TIMEOUT COUNTER
    MOV R6,00H
    MOV TH0,#-4
    MOV TL0,#00H
    JB OC8H,BB
    MOV ACC,#00H
    MOVX 1,@DPTR
    MOV 06,ACC
    CJNE A,#004H,CONT1 ;LOOK FOR END OF TEXT
    JMP EOTB
    CONT1: SETB ERX0
    CKR 0EEH
    INC DPTR
    MOV ACC,DPH
    JB ACC.2,EOTB
    SETB 0EEH
    JMP CONT
    EOTB: CLR EX0      ;END OF TEXT FOUND, DISABLE
                ;INT0
    SETB 0EEH
    SETB EA
    RETI
    BB: INC R7
    CJNE R7,#00H,CONT
    INC R6
    CJNE R6,#10H,CONT
    JMP TO
    CONT: SETB TRO
    SETB ET0
    SETB EA
    RETI
    TO: CLR OC9H
                ;SEND NEW STROBE PULSE IN
                ;RESPONSE TO TIMEOUT
    NOP
    NOP
    MOV R6,#00H
    MOV R7,#00H
    SETB OC9H
    JMP PSEND1
    ;RESET TO COUNTER
    ;END OF STROBE PULSE

```

Using the 87C451 microcontroller
in a Centronics printer buffer

DESCRIPTION
This application note describes the operation of the 87C451 microcontroller as a Centronics printer buffer. The 87C451 is a 16-bit microcontroller with 2Kbytes of on-chip ROM and 2Kbytes of on-chip RAM. It is designed to interface a personal computer and a printer. The 87C451 can only drive 3 LSTTL loads. It has 0-bit data stored into it by the IOSTB pin of the printer port. As the code size for this application is small (less than 1Kbytes), the on-chip instruction memory is sufficient. A program can be stored in the 87C451 using the 87C451 with a 4K x 8 EPROM. The 87C451 is designed to work with 1 Mbytes of DRAM. Although written with the 87C451 in mind, this design is applicable to the 80C451 and 80C451T.

Design Objectives
The objectives kept in mind during the design of this device were: provide a substantial size of buffer, keep the parts count and the power consumption to a minimum, and use readily available components.

A buffer size of 256K bytes was chosen because, although a 64K byte buffer is easily implemented using the 8081 family, a 64K external data storage capability is a little too small for today's printing applications. First, a page of text in graphics mode, using up many lines as many bytes as standard printing mode. Preserving a method for controlling 256K DRAMs shows off the NO capability of the 87C451, and it is very easy to add the extra address line for one megabit devices if a larger buffer is needed.

The 87C451 Microcontroller
The 87C451 is an 8-bit microcontroller based on the familiar 8081 family of devices. In fact, it is an 80C51 with three added ports: P4, P5, and P6. Port 4 and P5 (18 pins) are additional quasi-bidirectional I/O lines. Port 6 provides another 8 bits of I/O. Port 4 handles lines that can be programmed to operate in several useful modes for interfacing. The 87C451 comes in three versions: ROMless 80C451, 87C451 with 4K x 8 ROM, and 87C451 with 4K x 8 EPROM.

In this note, port 6 is used in the NO mode as a Centronics compatible printer output port. Additionally, the IOSTB and SFLA0 pins normally associated with port 6 are used as part of the input port logic. For a complete discussion of port 6 operating modes and programming, see the application note AN408 titled "87C451 Microcontroller Operation in Port 6".

Circuit Description
Figure 1 is a schematic diagram of the printer buffer circuit. Other than the 87C451 (U1) and the eight 256K DRAMs (U8-U15), only

256k Centronics printer buffer using the 87C451 microcontroller

AN417

DESCRIPTION

This application note describes a stand alone Centronics type parallel printer buffer using the 87C451 expanded I/O microcontroller. This type of unit would typically be placed between a personal computer and its printer. It captures the data to be printed at high speed, freeing the personal computer to go to other tasks, and sends data to the printer as required. As described here, 256k dynamic RAMs are used, providing over one quarter million characters of storage. If desired the design is easily modified to work with 1 megabit DRAMs. Although written with the 87C451 in mind, this design is applicable to the 80C451 and 83C451.

Design Objectives

The objectives kept in mind during the design of this device were: provide a substantial size of buffer, keep the parts count and the power consumption to a minimum, and use readily available components.

A buffer size of 256k bytes was chosen because, although a 64k byte buffer is very easily implemented using the 8051 family's 64k external data storage capabilities, it is a little too small for today's printing applications that print a page of text in graphics mode, using up twenty times as many bytes as standard printing mode. Presenting a method for controlling 256k DRAMs shows off the I/O capabilities of the 87C451, and it is very easy to add the extra address line for one megabit devices if a larger buffer is needed.

The 8XC451 Microcontroller

The 8XC451 is an 8-bit microcontroller based on the familiar 8051 family of devices. In fact, it is an 80C51 with three added ports: P4, P5, and P6. Ports 4 and 5 give 12 (16 in PLCC) additional quasi-bidirectional I/O lines. Port 6 provides another 8 bits of I/O, plus 4 handshake lines that can be programmed to operate in several useful modes for interfacing. The 8XC451 comes in three versions: ROMless 80C451, 83C451 with 4k x 8 ROM, and 87C451 with 4k x 8 EPROM.

In this note, port 6 is used in the I/O mode as a Centronics compatible printer output port. Additionally, the /IDS and BFLAG pins normally associated with port 6 are used as part of the input port logic. For a complete discussion of port 6 operating modes and programming, see the application note AN408 titled "83C451 Microcontroller Operation of Port 6."

Circuit Description

Figure 1 is a schematic diagram of the printer buffer circuit. Other than the 87C451 (U1), and the eight 256k DRAMs (U5-U12), only

two 74LS244 buffers (U2, U3) and a 76HCT374 (U4) octal flip-flop are needed. The U2 and U3 buffers are included to provide full drive capability for the output port and some of the handshake signals on the input port, as the output buffers on the 87C451 can only drive 3 LSTTL loads. U4 has 8-bit data strobed into it by the /STB pulse of the input port.

As the code size for this application is quite small (less than 1k bytes), the on-chip instruction memory is quite sufficient for program storage. For a production version, the 87C451 could be replaced with the 83C451 with a 4k x 8 masked ROM on chip. Note that port 0 and port 1 are not used in the present design; thus the 80C451 may be used in this application with the addition of an external address latch and EPROM.

The /RAS, /CAS, and /WR signals for the DRAM array are provided by port 3 bits /WR, /RD, and T1. Note that as in the 80C51, all port 3 signals are multifunctional. That is, each can be treated as a regular quasi-bidirectional port bit, or as having the special function indicated by its name. This feature is an advantage when using /WR and /RD as /RAS and /CAS control signals for a DRAM array. Treated as a normal port bit, the /WR pin is cleared and set by individual CLR and SETB instructions for a normal length RAM read or write cycle. However, when performing a refresh cycle, /RAS (port 3/WR) can be pulsed low using a dummy MOVX @R0,A (move to external data memory) instruction. This allows DRAM refresh to be done much more quickly than would otherwise be possible.

Port 1 and one bit from port 4 form the 9-bit address required when addressing the DRAM array. The data inputs to the array come from the parallel input data lines which are latched by U4. The RAM data outputs are fed to port 5. By making the data outputs available to the processor, it is possible to add some additional features to the firmware, such as control codes for printing multiple copies of a document, data compression, data conversion, etc. which are not implemented in this design.

Port 6 Operation

The /IDS (input data strobe) and BFLAG pins are normally used in conjunction with the port 6 bidirectional mode. In this mode, the /IDS pin is used to strobe data into the port 6 input latches, and BFLAG is used as flag output. In this application, however, these two bits are used to good effect as part of the (separate) input port logic. When a byte of data is strobed into U4 by the printer port of the host computer, the /STB signal connected to /IDS

sets the input buffer full flag (IBF). BFLAG is programmed to mirror the contents of IBF, and therefore becomes asserted. This makes it ideal to be used as the BUSY output for the input port. After the input port data has been read and stored in the RAM buffer, BFLAG is de-asserted by performing a dummy read of port 6, which clears IBF. To complete the input port logic, one of the port 3 pins, P3.4, is used as the acknowledge signal, and is asserted/de-asserted by software. The /ODS pin is tied to ground to permanently enable the port 6 output drivers. This does not cause difficulty as no data is being input into the port.

Note that programming port 6 to operate in the bidirectional mode as described above means the loss of /ODS as an acknowledge input. The acknowledge input is normally used to clear the OBF (output buffer full) flag, indicating that the printer is ready for another character. On the other hand, operating port 6 in the "output only" mode causes the loss of BFLAG as BUSY output. Because the input port requires an instant BUSY indication while the output port only needs to remember the occurrence of an acknowledge pulse, it makes sense to program port 6 to operate in the bidirectional mode, with /ODS grounded to enable the output drivers. The /INT1 pin can be used instead of /ODS to record the occurrence of an acknowledge pulse with the interrupt system.

Priority and Execution of Tasks

There are three tasks that must be performed in this system: Receive—servicing the input port and storing the input character; Transmit—sending stored characters to the output port as required; and Refresh—performing DRAM refresh. The timers and interrupt system are used to manage the execution and priority of these tasks. Figure 2 and Figure 3 illustrate the flow charts of these tasks. Firmware, broken into sections, performing these three functions as well as an initialization routine is provided.

The 51C256 DRAMs require a 256 row refresh every 4 milliseconds. Rather than do an entire refresh cycle every 4 milliseconds, it is done as 64 rows every millisecond. This leaves time for other tasks to get service "slices" more frequently. As DRAM refresh is obviously the highest priority, timer 0 is used as the refresh interval timer, and is programmed to the 16-bit mode, and set to the higher priority level in the interrupt priority (IP) register. The refresh code is written in-line rather than in a loop to maximize speed.

An interesting point to note is that when there are no characters stored, the DRAM does not need to be refreshed. If power consumption

256k Centronics printer buffer using the 87C451 microcontroller

AN417

is of concern, the 87C451 could be programmed to go into idle mode whenever the buffer were empty. A character strobed into the input port would cause an interrupt, restarting the 87C451; DRAM refresh would be maintained until the buffer was once again empty.

The next highest priority should be input port service, as the reason for having a printer buffer is to get the data out of the computer as quickly as possible. Therefore, the input port /STB signal is connected to the /INT0 pin (as well as U4's clock pin and /IDS). Interrupt 0 is programmed in the interrupt priority register to be at the lower interrupt level so it cannot prevent refresh service. The interrupt 0 service routine stores the input character at the next location in the DRAM array, using the technique of a circular FIFO buffer. The routine also sends back an acknowledge pulse by clearing and setting the P3.4 pin, and then clears the BUSY (BFLAG) pin by performing a dummy read of port 6 (unless this character caused the buffer to be completely full).

During periods of access to the DRAM array by the input and output routines, the global interrupt enable bit (EA) is cleared so that the refresh interrupt does not disturb the contents of ports 1 and 4, or the /RAS, /CAS, and /WR signals.

The printer (output port) service routine runs all the time, except when the CPU is called to service the other conditions, therefore having the lowest priority. If there are characters in the buffer, polling is used to check for output port BUSY status. If the printer is not busy, then the character is sent, and the output port /STB pin (P4.3) is cleared and set. The output port /ACK line is connected to the /INT1 pin, so that the negative going edge of the /ACK signal is recorded as an interrupt pending. A very short INT1 service routine sets a software flag to indicate that the printer acknowledge the last character.

Possible Enhancements

There are a number of features that could be added to this design. As mentioned previously, the microcontroller could be put into the idle mode when the buffer is empty, conserving power.

The software could be enhanced to provide features such as multiple copies of a document, data compression, data conversion, automatic printer setup, etc. The PC operating system could be suitably modified to send a header for each file to be printed, containing these parameters. There is plenty of room for operating firmware expansion, and plenty of horsepower left in the 87C451 to handle these features.

The two serial port pins RxD and TxD were deliberately left unused so that input and/or

output ports are easily implemented for serial interfaces or printers using the built-in UART. The pins used for parallel port handshaking could then be used as serial handshaking lines, providing the standard "modem" signals.

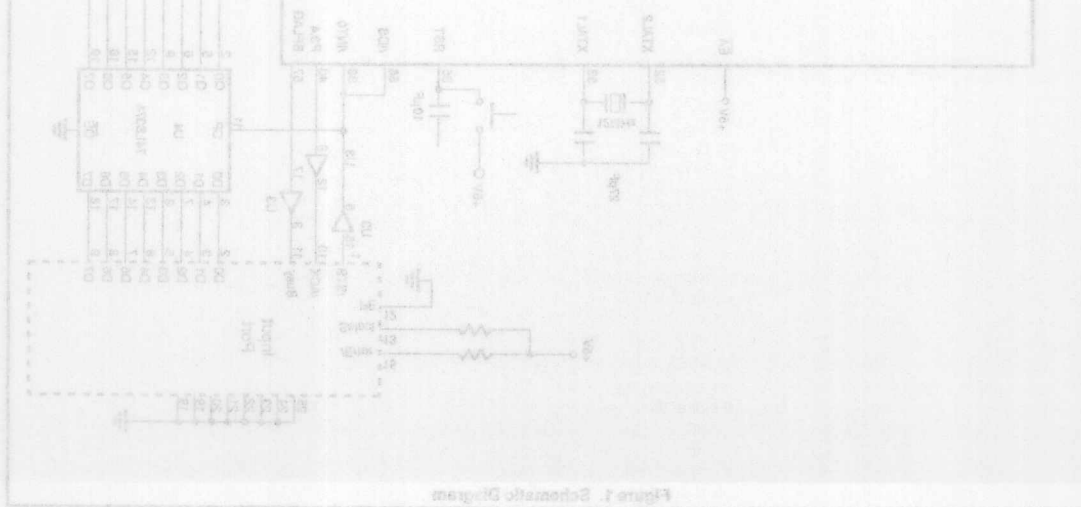
Combining the above two features, this circuit could act as a "splitter." By connecting a daisy-wheel printer to the serial port, a dot-matrix printer to the parallel port, and sending an "address" flag in the file header, simultaneous letter-quality and draft printing could be done.

The size of the DRAM array is easily expanded to one megabyte or large devices by connecting the additional address pins to port 4 bits 1 and 2. Only slight modifications to the operating firmware would be required.

Conclusion

The SC8XC451 microcontrollers provide plenty of I/O pins that previously had to be implemented by clumsy I/O expansion methods. The flexibility of port 6 means that this device can be used in a wide variety of applications requiring special port functions, while still using the industry standard 8051 instruction set.

The Application Note, describing a typical parallel printer buffer, makes full use of the 8XC451 features, yet allows room for enhancement and expansion.



AN417



256k Centronics printer buffer using the 87C451 microcontroller

AN417

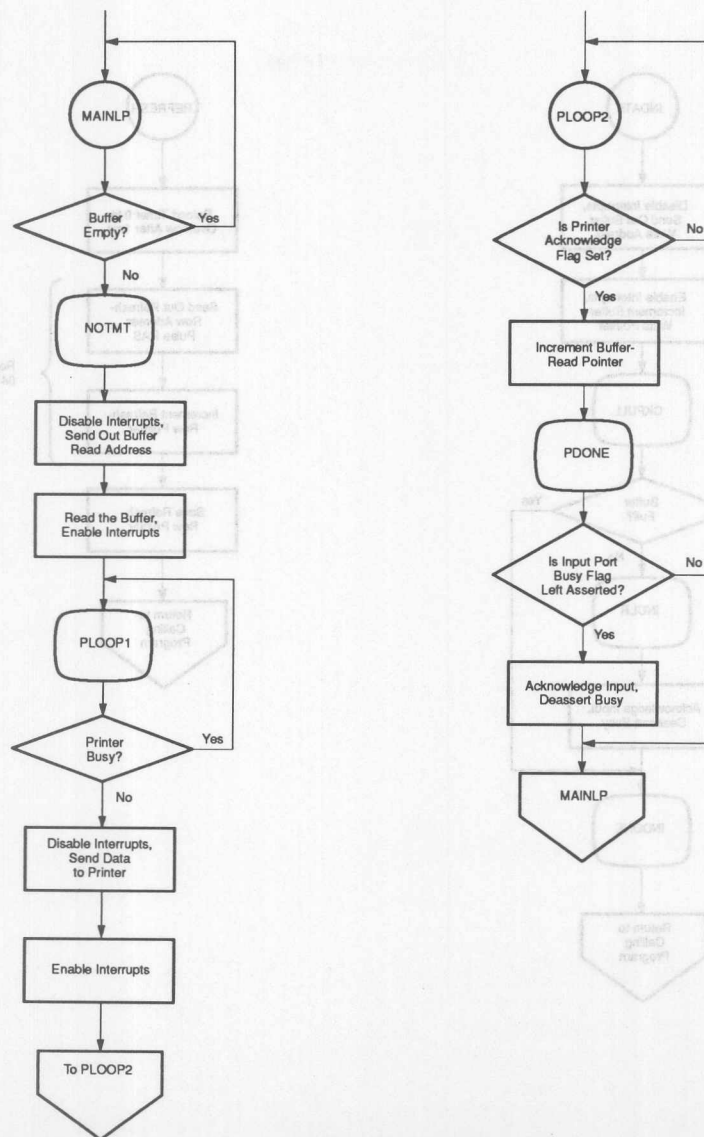


Figure 2. Flowchart of Transmit Operation

256k Centronics printer buffer using the 87C451 microcontroller

AN417

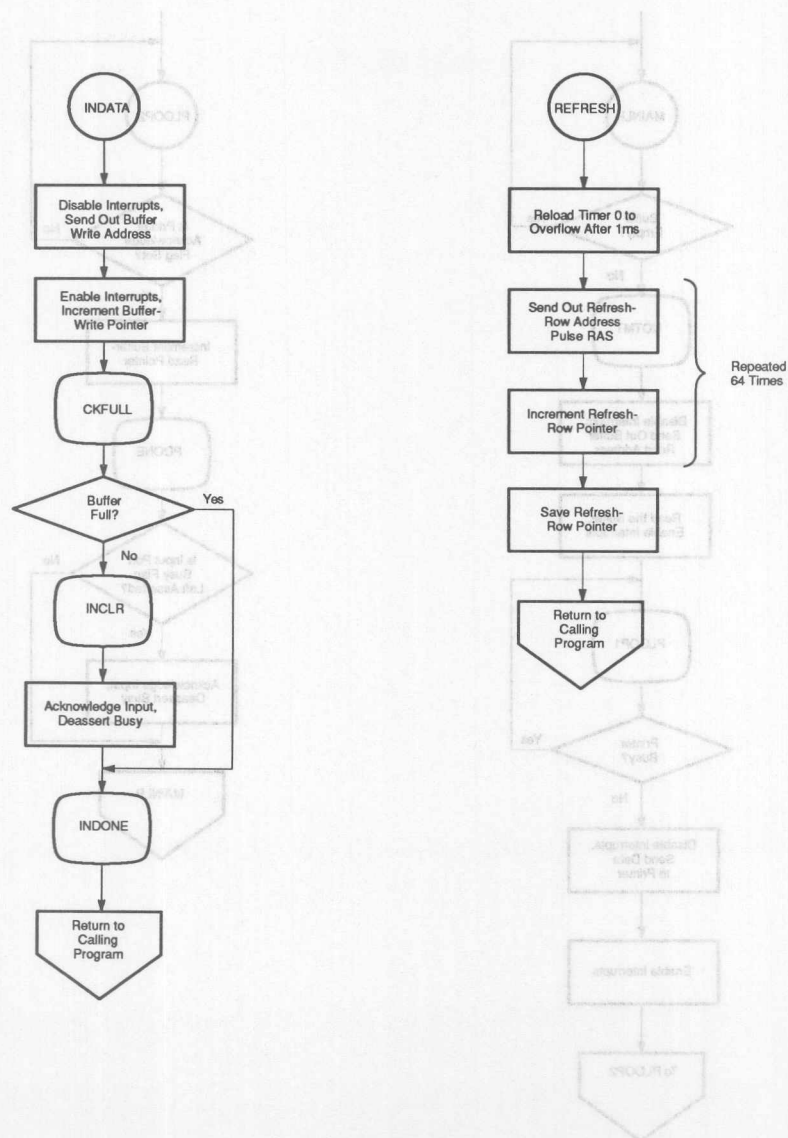


Figure 3. Flowchart of Receive and Refresh Operation

256k Centronics printer buffer using the 87C451 microcontroller

AN417

```
XMIT:      MOV DPTR,8001H
TEST:      MOVX A,@DPTR      ;READ THE CSR
           JB ACC.0,TEST      ;TEST IBF FLAG
```

256K PRINTER BUFFER PROGRAM USING THE 8xC451 FOR CENTRONICS PARALLEL PRINTER PORTS

PHILIPS SEMICONDUCTORS
OCTOBER, 1988

\$Mod451

\$Title(*XC451 Printer Buffer)

\$Date(10/28/88)

PORT USAGE:

P0 Not used (reserved for data/address bus when external program memory is used).

P1 Lower 8 bits of DRAM address (A0 – A7).

P2 Not used (reserved for high-order address bus when external program memory is used).

P3.0 (Reserved for serial port.)

P3.1 (Reserved for serial port.)

P3.2 (/INT0) Input port strobe input (interrupt).

P3.3 (/INT1) Output port acknowledge input (interrupt).

P3.4 Input port acknowledge output.

P3.5 DRAM write enable output.

P3.6 (/WR) DRAM row address select output.

P3.7 (/RD) DRAM column address select output.

P4.0 Upper bit of DRAM address (A8).

P4.1 Reserved as an extra address line for 1 megabit DRAMS.

P4.2 Not used.

P4.3 Output port busy input (OBUSY).

P4.4–P4.7 Unused (not available on 64-pin DIP package).

P5 DRAM output data.

P6 Parallel output port.

/DS Input port strobe input (ISTB).

BFLAG Input port busy output (IBUSY).

AFLAG Output port strobe output (OSTB).

/ODS Port 6 output enable, tied low.

Internal Register/RAM Usage:

```
REFCNT EQU 020h      ; Low order refresh byte;
```

The following refer to the circular FIFO buffer
implemented in the DRAM array.

```
INLOW EQU 22h      ; Incoming address low byte.
INMID EQU 23h      ; Incoming address mid byte.
INHI EQU 24h       ; Incoming address high byte.
OUTLOW EQU 25h     ; Outgoing address low byte.
OUTMID EQU 26h     ; Outgoing address mid byte.
OUTH EQU 27h       ; Outgoing address high byte.

OACK EQU 28h       ; Holds flag for output port acknowledge.
FOACK BIT OACK.0   ; Bit-address of output port acknowledge flag.
```

256k Centronics printer buffer using the 87C451 microcontroller

AN417

Miscellaneous Equates:

```

TIME      EQU      -1000      ; Value for 1000 timer clocks = 1 millisecond.
TIMEHI    EQU      HIGH TIME  ; High byte of timer value.
TIMELO    EQU      LOW TIME   ; Low byte of timer value.
RAS       BIT      P3.6       ; DRAM column address select.
CAS       BIT      P3.7       ; DRAM row address select.
DRAMWR    BIT      P3.5       ; DRAM write control line.
IACK      BIT      P3.4       ; Input port ACK output.
ISTB      BIT      P3.2       ; Input port strobe line (INT0).
OBUSY     BIT      P4.3       ; Output port BUSY input.
OSTB      BIT      MA0        ; Output port strobe (MA0 bit in port 6 CSR).

```

Reset and Interrupt Jump Table

```

ORG 00h      ; Power-on reset.
AJMP START

ORG 03h      ; INT 0.
AJMP INDATA  ; Data at input port.

ORG 0Bh      ; Timer 0.
AJMP REFRESH ; Refresh DRAM array.

ORG 13h      ; INT 1.
AJMP OPACK   ; Output port acknowledge.

```

Power up reset routine:

Set up refresh timer, enable timer interrupt and external interrupt, initialize circular buffer pointers.

```

START:  ORG 18h      ; Initialize stack pointer.
        MOV SP,#40h
        MOV A,#00    ; Initialize refresh counter.
        MOV REFCNT,A
        MOV INLOW,A   ; Initialize FIFO pointers.
        MOV INMID,A
        MOV INHI,A
        MOV OUTLOW,A
        MOV OUTMID,A
        MOV OUTHI,A

; Initialize interrupt priority register so that DRAM refresh
; (TF0) gets high priority, input port service (IE0) and output
; port acknowledge service get lower priority. All other
; interrupts set to lower priority level.

        MOV IE,#00000111b ; Timer0, INT0, and INT1 enabled.
        MOV IP,#00000010b ; Timer0 high priority.
        MOV TLO,#TIMELO
        MOV TH0,#TIMEHI
        MOV TMOD,#00000001b ; Operate Timer0 in mode 1.
        MOV TCON,#00010101b ; Timer0 run, I0 and I1 = edge.

```

256k Centronics printer buffer using the 87C451 microcontroller

AN417

```

; Initialize Port 6 Control and Status Register.
; - 'BFLAG' mode set to output value of IBF
;   (input port BUSY signal : IBUSY)
; - 'AFLAG' set as logic 1 output
;   (output port strobe signal : OSTB)
; - 'IDS' active on negative level
;   (input port strobe signal : ISTB)
; MOV     CSR,#10011100b
; MOV     A,P6          ; Dummy read of P6 to clear IBF (IBUSY).
; SETB    EA            ; Enable interrupts.

; *****

; Main Routine:
; Executes while not performing DRAM refresh or servicing
; input port interrupt.

; Check if buffer is not empty by comparing input and output
; pointers. If not empty, go to NOTMT to output a byte.

MAINLP:  MOV     A,INLOW      ; Compare pointers.
         CJNE    A,OUTLOW,NOTMT
         MOV     A,INMID
         CJNE    A,OUTMID,NOTMT
         MOV     A,INHI
         CJNE    A,OUTH,NOTMT
         SJMP    MAINLP

; Buffer is not empty: compute row & column addresses for
; a read cycle from DRAM.

NOTMT:   MOV     R4,OUTLOW    ; Save low byte of row.
         MOV     R5,OUTMID    ; Save upper bit of row.
         MOV     A,OUTH       ; Shift to align correctly.
         RRC     A
         MOV     R7,A          ; Save upper column bit.
         MOV     A,OUTMID     ; Get low byte of column.
         RRC     A            ; Shift in bit from OUTH.
         MOV     R6,A          ; Save.

; Now do actual DRAM access to get the data byte at computed
; address. Disable interrupts so we don't lose what we put
; out on the ports.

         CLR     EA            ; Disable interrupts.
         MOV     P1,R4         ; Low byte row address.
         MOV     A,R5         ; Get high byte row address.
         ORL     A,#0FEh       ; Make sure OBUSY stays high.
         MOV     P4,A
         CLR     RAS           ; /RAS low.
         MOV     P1,R6         ; Low byte column address.
         MOV     A,R7         ; High byte column address.
         ORL     A,#0FEh       ; Make sure OBUSY stays high.
         MOV     P4,A
         CLR     CAS           ; /CAS low.
         MOV     R4,P5         ; Get the data byte

         SETB    CAS           ; /CAS high.
         SETB    RAS           ; /RAS high.
         CLR     FOACK         ; Clear acknowledge flag.
         SETB    EA            ; Re-enable interrupts.

```

256K Centronics printer buffer using the 87C451 microcontroller

AN417

```

PLOOP1:  JB      OBUSY,PLOOP1    ; Loop if printer busy.
;
;      CLR      EA              ; Disable interrupts.
;      MOV      P6,R4           ; Move byte to output port.
;      CLR      MA0             ; Assert output port strobe.
;      NOP                     ; Kill some time.
;      NOP
;      NOP
;      NOP
;      SETB     MA0             ; De-assert output port strobe.
;      SETB     EA              ; Re-enable interrupts.
;
;      Following waits for /ACK to occur on output port. Loops on
;      acknowledge flag which is set by INT1 service routine when
;      /ACK occurs.
PLOOP2:  JNB      FOACK,PLOOP2    ; Wait till /ACK occurs.
;
;      INC      OUTLOW          ; Increment output buffer pointer.
;      MOV      A,OUTLOW
;      CJNE     A,#00,PDONE
;      INC      OUTMID
;      MOV      A,OUTMID
;      CJNE     A,#00,PDONE
;      MOV      A,OUTH
;      INC      A
;      ANL      A,#03h          ; Eliminate unused address bits
;      MOV      OUTH,A          ; and save.
;
;      Check if input port busy flag was left asserted, indicating that
;      the buffer was full after last input. If so, acknowledge input
;      port and de-assert input busy signal.
PDONE:   JNB      IBF,MAINLP      ; Not busy, return to main loop.
;      CLR      EA              ; Disable interrupts.
;      CLR      IACK            ; Assert /IACK.
;      NOP                     ; Wait 7 microseconds.
;      NOP
;      NOP
;      NOP
;      NOP
;      MOV      A,P6            ; Dummy read of P6 clears IBF (IBUSY).
;      NOP                     ; Wait 5 microseconds.
;      NOP
;      NOP
;      NOP
;      SETB     IACK            ; De-assert /IACK.
;      SETB     EA              ; Re-enable interrupts.
;      AJMP     MAINLP          ; Return to main loop.

```


256k Centronics printer buffer using the 87C451 microcontroller

AN417

.....				13:	A,0R@	XVOM
:				14:	P1	INC
:				15:	A,0R@	XVOM
:				16:	P1	INC
:				17:	A,0R@	XVOM
:				18:	P1	INC
:				19:	A,0R@	XVOM
:				20:	P1	INC
:				21:	A,0R@	XVOM
:				22:	P1	INC
:				23:	A,0R@	XVOM
:				24:	P1	INC
:				25:	A,0R@	XVOM
:				26:	P1	INC
:				27:	A,0R@	XVOM
:				28:	P1	INC
:				29:	A,0R@	XVOM
:				30:	P1	INC
:				31:	A,0R@	XVOM
:				32:	P1	INC
:				33:	A,0R@	XVOM
:				34:	P1	INC
:				35:	A,0R@	XVOM
:				36:	P1	INC
:				37:	A,0R@	XVOM
:				38:	P1	INC
:				39:	A,0R@	XVOM
:				40:	P1	INC
:				41:	A,0R@	XVOM
:				42:	P1	INC
:				43:	A,0R@	XVOM
:				44:	P1	INC
:				45:	A,0R@	XVOM
:				46:	P1	INC
:				47:	A,0R@	XVOM
:				48:	P1	INC
:				49:	A,0R@	XVOM
:				50:	P1	INC
:				51:	A,0R@	XVOM
:				52:	P1	INC
:				53:	A,0R@	XVOM
:				54:	P1	INC
:				55:	A,0R@	XVOM
:				56:	P1	INC
:				57:	A,0R@	XVOM
:				58:	P1	INC
:				59:	A,0R@	XVOM
:				60:	P1	INC
:				61:	A,0R@	XVOM
:				62:	P1	INC
:				63:	A,0R@	XVOM
:				64:	P1	INC
:				65:	A,0R@	XVOM
:				66:	P1	INC
:				67:	A,0R@	XVOM
:				68:	P1	INC
:				69:	A,0R@	XVOM
:				70:	P1	INC
:				71:	A,0R@	XVOM
:				72:	P1	INC
:				73:	A,0R@	XVOM
:				74:	P1	INC
:				75:	A,0R@	XVOM
:				76:	P1	INC
:				77:	A,0R@	XVOM
:				78:	P1	INC
:				79:	A,0R@	XVOM
:				80:	P1	INC
:				81:	A,0R@	XVOM
:				82:	P1	INC
:				83:	A,0R@	XVOM
:				84:	P1	INC
:				85:	A,0R@	XVOM
:				86:	P1	INC
:				87:	A,0R@	XVOM
:				88:	P1	INC
:				89:	A,0R@	XVOM
:				90:	P1	INC
:				91:	A,0R@	XVOM
:				92:	P1	INC
:				93:	A,0R@	XVOM
:				94:	P1	INC
:				95:	A,0R@	XVOM
:				96:	P1	INC
:				97:	A,0R@	XVOM
:				98:	P1	INC
:				99:	A,0R@	XVOM
:				100:	P1	INC
:				101:	A,0R@	XVOM
:				102:	P1	INC
:				103:	A,0R@	XVOM
:				104:	P1	INC
:				105:	A,0R@	XVOM
:				106:	P1	INC
:				107:	A,0R@	XVOM
:				108:	P1	INC
:				109:	A,0R@	XVOM
:				110:	P1	INC
:				111:	A,0R@	XVOM
:				112:	P1	INC
:				113:	A,0R@	XVOM
:				114:	P1	INC
:				115:	A,0R@	XVOM
:				116:	P1	INC
:				117:	A,0R@	XVOM
:				118:	P1	INC
:				119:	A,0R@	XVOM
:				120:	P1	INC
:				121:	A,0R@	XVOM
:				122:	P1	INC
:				123:	A,0R@	XVOM
:				124:	P1	INC
:				125:	A,0R@	XVOM
:				126:	P1	INC
:				127:	A,0R@	XVOM
:				128:	P1	INC
:				129:	A,0R@	XVOM
:				130:	P1	INC
:				131:	A,0R@	XVOM
:				132:	P1	INC
:				133:	A,0R@	XVOM
:				134:	P1	INC
:				135:	A,0R@	XVOM
:				136:	P1	INC
:				137:	A,0R@	XVOM
:				138:	P1	INC
:				139:	A,0R@	XVOM
:				140:	P1	INC
:				141:	A,0R@	XVOM
:				142:	P1	INC
:				143:	A,0R@	XVOM
:				144:	P1	INC
:				145:	A,0R@	XVOM
:				146:	P1	INC
:				147:	A,0R@	XVOM
:				148:	P1	INC
:				149:	A,0R@	XVOM
:				150:	P1	INC
:				151:	A,0R@	XVOM
:				152:	P1	INC
:				153:	A,0R@	XVOM
:				154:	P1	INC
:				155:	A,0R@	XVOM
:				156:	P1	INC
:				157:	A,0R@	XVOM
:				158:	P1	INC
:				159:	A,0R@	XVOM
:				160:	P1	INC
:				161:	A,0R@	XVOM
:				162:	P1	INC
:				163:	A,0R@	XVOM
:				164:	P1	INC
:				165:	A,0R@	XVOM
:				166:	P1	INC
:				167:	A,0R@	XVOM
:				168:	P1	INC
:				169:	A,0R@	XVOM
:				170:	P1	INC
:				171:	A,0R@	XVOM
:				172:	P1	INC
:				173:	A,0R@	XVOM
:				174:	P1	INC
:				175:	A,0R@	XVOM
:				176:	P1	INC
:				177:	A,0R@	XVOM
:				178:	P1	INC
:				179:	A,0R@	XVOM
:				180:	P1	INC
:				181:	A,0R@	XVOM
:				182:	P1	INC
:				183:	A,0R@	XVOM
:				184:	P1	INC
:				185:	A,0R@	XVOM
:				186:	P1	INC
:				187:	A,0R@	XVOM
:				188:	P1	INC
:				189:	A,0R@	XVOM
:				190:	P1	INC
:				191:	A,0R@	XVOM
:				192:	P1	INC
:				193:	A,0R@	XVOM
:				194:	P1	INC
:				195:	A,0R@	XVOM
:				196:	P1	INC
:				197:	A,0R@	XVOM
:				198:	P1	INC

256k Centronics printer buffer using the 87C451 microcontroller

AN417

```

MOVX @R0,A ; 21
INC P1
MOVX @R0,A ; 22
INC P1
MOVX @R0,A ; 23
INC P1
MOVX @R0,A ; 24
INC P1
MOVX @R0,A ; 25
INC P1
MOVX @R0,A ; 26
INC P1
MOVX @R0,A ; 27
INC P1
MOVX @R0,A ; 28
INC P1
MOVX @R0,A ; 29
INC P1
MOVX @R0,A ; 30
INC P1
MOVX @R0,A ; 31
INC P1
MOVX @R0,A ; 32
INC P1
MOVX @R0,A ; 33
INC P1
MOVX @R0,A ; 34
INC P1
MOVX @R0,A ; 35
INC P1
MOVX @R0,A ; 36
INC P1
MOVX @R0,A ; 37
INC P1
MOVX @R0,A ; 38
INC P1
MOVX @R0,A ; 39
INC P1
MOVX @R0,A ; 40
INC P1
MOVX @R0,A ; 41
INC P1
MOVX @R0,A ; 42
INC P1
MOVX @R0,A ; 43
INC P1
MOVX @R0,A ; 44
INC P1
MOVX @R0,A ; 45
INC P1
MOVX @R0,A ; 46
INC P1
MOVX @R0,A ; 47
INC P1
MOVX @R0,A ; 48
INC P1
MOVX @R0,A ; 49
INC P1
MOVX @R0,A ; 50
INC P1
MOVX @R0,A ; 51
INC P1
MOVX @R0,A ; 52
INC P1
MOVX @R0,A ; 53
INC P1
MOVX @R0,A ; 54
INC P1

```

256k Centronics printer buffer using the 87C451 microcontroller

AN417

```

MOVX @R0,A      ;55
INC P1
MOVX @R0,A      ;56
INC P1
MOVX @R0,A      ;57
INC P1
MOVX @R0,A      ;58
INC P1
MOVX @R0,A      ;59
INC P1
MOVX @R0,A      ;60
INC P1
MOVX @R0,A      ;61
INC P1
MOVX @R0,A      ;62
INC P1
MOVX @R0,A      ;63
INC P1

INC P1           ; Adjust for next time
MOV REFCNT,P1    ; and save.
POP PSW
RETI

```

Data at Input Port:

This routine is called via interrupt INT0 whenever data is strobed into the input port. It saves the data into the DRAM array and increments the input pointer. If the output pointer is now equal to the input pointer, then the buffer is full, and we leave the busy flag set so that no more data can be input until some is output and the buffer is no longer full.

```

INDATA: PUSH PSW
        PUSH ACC
        MOV R1,INLOW      ; Lower 8 bits of row to R1.
        MOV R2,INMID      ; Upper bit of row to R2.
        MOV A,INHI        ; Get upper 2 bits.
        RRC A             ; LSB to carry.
        MOV R0,A
        MOV A,INMID
        RRC A             ; Shift bit into MSB.
        MOV R3,A          ; Save.

        CLR EA           ; Disable interrupts.
        MOV P1,R1        ; LSB Row address.
        MOV A,R2         ; MSB row address.
        ORL A,#0FEh      ; Make sure OBUSY stays high.
        MOV P4,A         ; MSB row address.
STBLP:  JNB ISTR,STBLP    ; Check for end of strobe before DRAM write.
        CLR RAS          ; /RAS low.
        CLR DRAMWR       ; /WR low.
        MOV P1,R3        ; LSB column address.
        MOV 1,R0         ; MSB column address.
        ORL A,#0FEh      ; Make sure OBUSY stays high.
        MOV P4,A         ; MSB column address.
        MOVX A,@R0       ; Pulse /CAS low.
        SETB RAS         ; /RAS high.
        SETB DRAMWR      ; /WR high.
        SETB EA          ; Re-enable interrupts.

```

256k Centronics printer buffer using the 87C451 microcontroller

AN417

```

INC     INLOW      ; Increment input buffer pointer.
MOV     A,INLOW
CJNE    A,#00,CKFULL
INC     INMID
MOV     A,INMID
CJNE    A,#00,CKFULL
MOV     A,INHI
INC     A
ANL     A,#03h      ; Eliminate unused address bits.
MOV     INHI,A

```

```

;
; Compare input pointer to output pointer to see if the buffer is full.
;

```

```

CKFULL: MOV     A,INLOW
CJNE    A,OUTLOW,INCLR
MOV     A,INMID
CJNE    A,OUTMID,INCLR
MOV     A,INHI
CJNE    A,OUTHI,INCLR

```

```

;
; If we get here, the buffer is full, so skip the acknowledge pulse.
;

```

```

        SJMP     INDONE

```

```

;
; Send acknowledge pulse on /IACK line for 7 microseconds,
; de-assert input BUSY signal halfway through.
;

```

```

INCLR:  CLR     EA      ; Disable interrupts.
        CLR     IACK    ; Assert /IACK.
        NOP     ; Wait 7 microseconds.
        NOP
        NOP
        NOP
        NOP
        NOP
        MOV     A,P6     ; Dummy read of P6 clears IBF (IBUSY).
        NOP
        POP     ACC      ; Wait 5 microseconds before clearing /IACK.
        POP     PSW
        SETB    IACK     ; De-assert /IACK.
        SETB    EA      ; Re-enable interrupts.
        RETI

```

```

END

```

Counter/timer 2 of the 83C552 microcontroller AN418

INTRODUCTION TO THE 83C552

The 83C552 is an 80C51 derivative with several extended features: 8k ROM, 256 bytes RAM, 10-bit A/D converter, two PWM channels, two serial I/O channels, six 8-bit I/O ports, and four counter/timers. The architecture of the 83C552 is identical to that of the 80C51, making the two devices fully code compatible. The additional peripheral functions are added to the 80C51 Special Function Register space, and the interrupt structure is modified accordingly. This information is detailed in other references on the 83C552. The focus of this application note is on one of the timers of the 83C552, Counter/Timer 2.

This counter/timer includes capture, compare, and high-speed output capabilities which facilitate many control oriented tasks. The objective of this note is to make users of the 83C552 aware of this counter/timer subsystem and assist the use of this subsystem by a detailed explanation of its operation supported by actual application examples.

TIMER 2 OF THE 83C552

Timer 2 of the 83C552 is in fact a timing controller and has an associated programmable array. The Timer 2 subsystem consists of three parts:

1. The time base consists of a 16-bit timer with a 3-bit prescaler. The master clock for the subsystem can be derived from the on-chip oscillator (fosc) or an external input, T2. It has an external reset, RT2, by which a signal applied to this input can reset the timer if the external reset is enabled.
2. A capture system consisting of four capture registers and four capture inputs which can be used for a wide variety of time measurements on external signals.
3. A compare system consisting of three compare registers and eight associated high-speed outputs which can be activated upon a match between the 16-bit timer and one of the compare registers.

For reference a complete block diagram of the 83C552 Counter/Timer 2 subsystem is shown in Figure 1.

16-BIT COUNTER/TIMER

The description of Counter/Timer 2 in the following paragraphs is intended to be a general overview. Details on architecture, address locations, interrupt structure, and timer operation are given in the 83C552 Users Manual. This users manual may be useful to complement the material presented in this application note. References to registers, bits, I/O ports, and on-chip hardware will relate directly to 83C552 Users Manual nomenclature. This application note will focus on the use of Counter/Timer 2 as a powerful input capture and high-speed output facilitator through some specific examples and not on the detailed coding.

The counter/timer consists of a 16-bit counter which is readable by software through special function registers TM2L and TM2H. The timer itself has two overflow flags, one after the entire 16-bit counter and one attached to the eighth stage. This latter flag reflects an overflow from the first byte of the counter. These two flags are present in register TM2IR and are labeled T2BO for the overflow from the first byte and T2OV for the overflow from the entire 16 bits. These flags may be used to generate an interrupt.

The counter timer is controlled directly through the special function register TM2CON, the timer 2 control register. This register also contains certain status flags.

The prescaler divides the input clock by a programmable ratio. The prescaler divide value is programmable to divide by 1, 2, 4, or 8 as controlled by T2PO and T2P1 in TM2CON.

The input clock to the prescaler is either fosc/12 or the external input, T2. The clock input to the prescaler may also be shut off. This clock input selection is controlled by bits T2MS0 and T2MS1 in TM2CON.

If T2 is used as the input clock to the timer 2 subsystem, the hardware logic samples this input and looks for a low-to-high transition. If the logic detects a logic 0 at the T2 input in state S2P1 of the microcontroller and a logic 1 in state S5P1, then this is recognized as a low-to-high transition, and the prescaler is incremented. The prescaler is incremented in the second cycle after the cycle in which the transition was detected. If the transition is detected before S2P1 is finished, the prescaler is incremented in the next cycle. This timing is shown in Figure 2. Note that this sampling rate is twice that of the normal 80C51 timers, T0 and T1; therefore T2 has

twice the maximum external counting rate as compared to the standard timers.

Any programming of the clock source or the prescaler divide ratio results in a reset of the prescaler. This allows the state of the timer subsystem to be in a known state upon programming. The main 16-bit timer cannot be reset by software but it is reset by activating the reset pin or using the external reset, RT2. The external reset, RT2, can be enabled or disabled by bit T2ER in TM2CON. These resets reset the prescaler as well as the 16-bit counter.

Only one interrupt is available from the 16-bit counter timer. Two bits in TM2CON control whether TM2L, TM2H, or both flags will be used to generate the interrupt. A selection for no interrupt is also possible.

Capture System

The capture system is a powerful tool to measure the width of pulses or repetition rates. There are four independent inputs for the signals to be analyzed, CTI0 through CTI3. These inputs are alternate functions to port 1. Each input is connected to a dedicated capture register. A transition at any of these inputs will cause the content of the 16-bit counter/timer to be loaded into the respective capture register. The capture can occur upon various conditions of the input signal as specified by certain bits in the capture control register, CTCON. Each input can be set to cause a capture on a low-to-high transition, a high-to-low transition, or on both transitions. Upon a capture taking place, each input causes an interrupt flag to be set in the Timer 2 Interrupt Flag Register, TM2IR. If enabled, an interrupt will be generated.

One of the capture inputs is shown in more detail in Figure 3. All of the other capture inputs are similar to this one. The capture input is gated with the capture enable bits CTN0 and CTP0, which are located in CTCON. According to the status of these bits, the desired edges are selected to generate the capture enable pulse. The input pulse transient detection is at the input of the enable pulse generator. The input signal is sampled at S1P1 of the machine cycle. If a logic 1 is detected when a logic 0 was detected at the same time in the previous cycle, then the event is taken as a transition. An enable pulse is sent to the capture register, and the contents of timer 2 is copied into the capture register at the end of this machine cycle. The interrupt flag CTI0 is also set.

Counter/timer 2 of the 83C552 microcontroller AN418

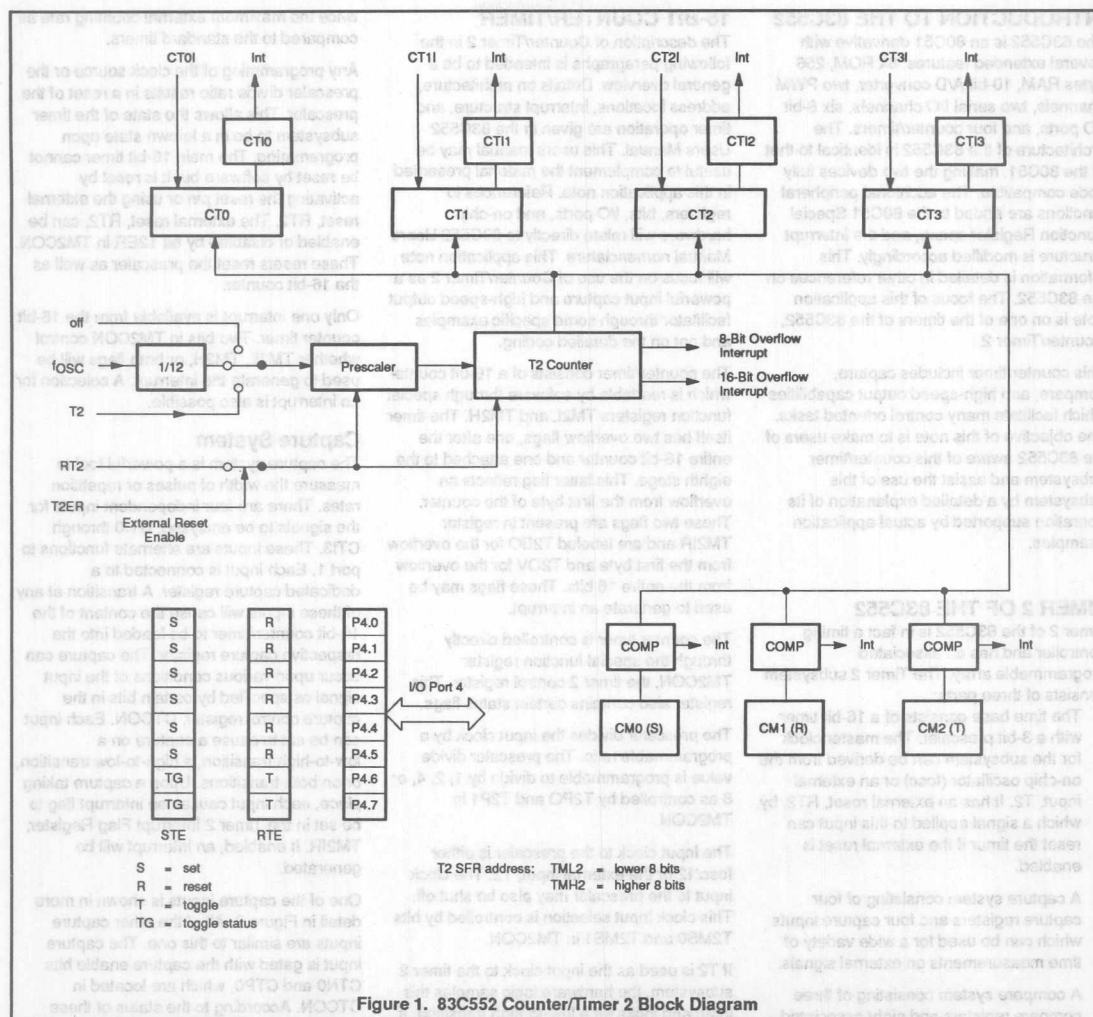
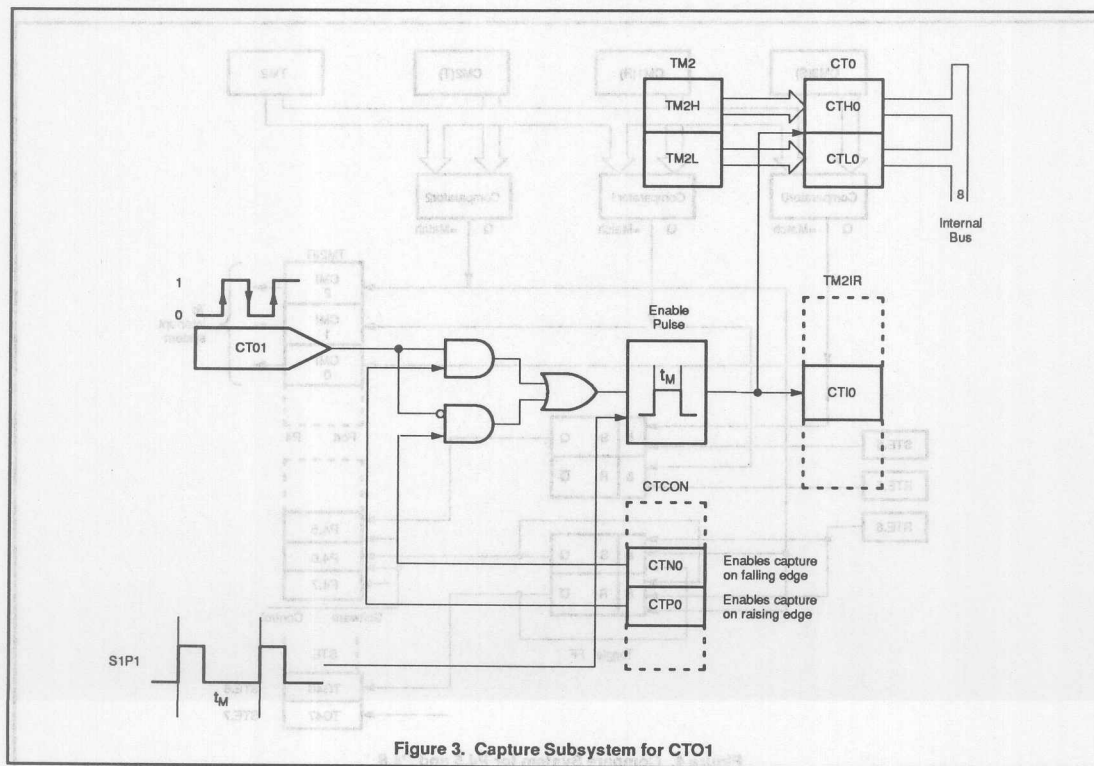
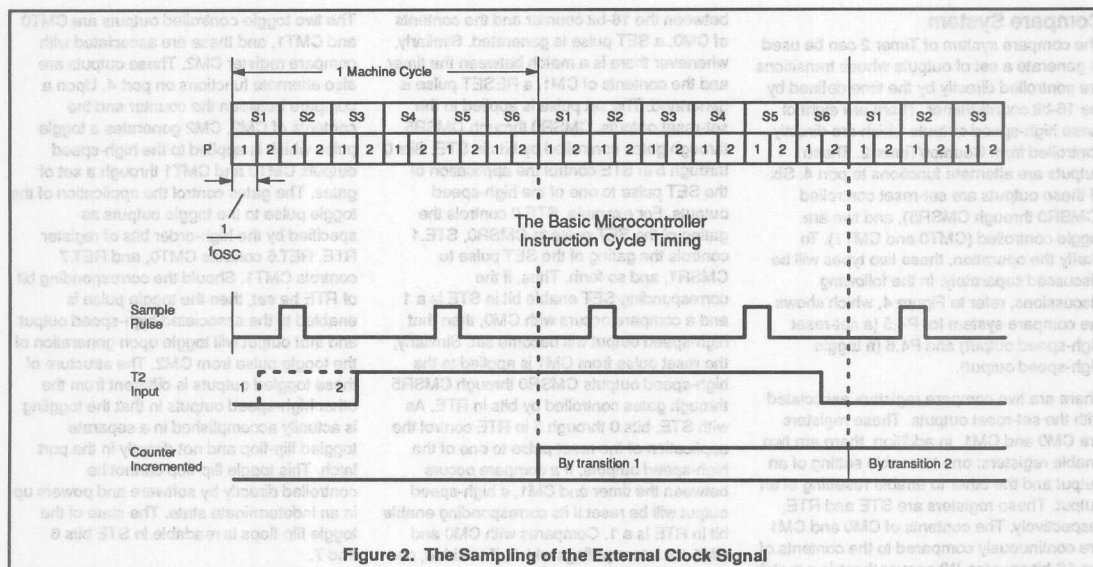


Figure 1. 83C552 Counter/Timer 2 Block Diagram

Counter/timer 2 of the 83C552 microcontroller AN418



Counter/timer 2 of the 83C552 microcontroller AN418

Compare System

The compare system of Timer 2 can be used to generate a set of outputs whose transitions are controlled directly by the time defined by the 16-bit counter/timer. There are eight of these high-speed outputs which are directly controlled from Counter/Timer 2. These outputs are alternate functions to port 4. Six of these outputs are set-reset controlled (CMSR0 through CMSR5), and two are toggle controlled (CMT0 and CMT1). To clarify the operation, these two types will be discussed separately. In the following discussions, refer to Figure 4, which shows the compare system for P4.5 (a set-reset high-speed output) and P4.6 (a toggle high-speed output).

There are two compare registers associated with the set-reset outputs. These registers are CM0 and CM1. In addition, there are two enable registers: one to enable setting of an output and the other to enable resetting of an output. These registers are STE and RTE, respectively. The contents of CM0 and CM1 are continuously compared to the contents of the 16-bit counter. Whenever there is a match

between the 16-bit counter and the contents of CM0, a SET pulse is generated. Similarly, whenever there is a match between the timer and the contents of CM1, a RESET pulse is generated. The set pulse is applied to the set-reset outputs, CMSR0 through CMSR5, through gates controlled by bits in STE. Bits 0 through 5 in STE control the application of the SET pulse to one of the high-speed outputs. For example, STE.0 controls the gating of the SET pulse to CMSR0, STE.1 controls the gating of the SET pulse to CMSR1, and so forth. Thus, if the corresponding SET enable bit in STE is a 1 and a compare occurs with CM0, then that high-speed output will become set. Similarly, the reset pulse from CM1 is applied to the high-speed outputs CMSR0 through CMSR5 through gates controlled by bits in RTE. As with STE, bits 0 through 5 in RTE control the application of the reset pulse to one of the high-speed outputs. If a compare occurs between the timer and CM1, a high-speed output will be reset if its corresponding enable bit in RTE is a 1. Compares with CM0 and CM1 set interrupt flags which, if enabled, can be used to generate an interrupt.

The two toggle-controlled outputs are CMT0 and CMT1, and these are associated with compare register CM2. These outputs are also alternate functions on port 4. Upon a compare between the counter and the contents of CM2, CM2 generates a toggle pulse which is applied to the high-speed outputs CMT0 and CMT1 through a set of gates. The gates control the application of the toggle pulse to the toggle outputs as specified by the high-order bits of register RTE. RET.6 controls CMT0, and RET.7 controls CMT1. Should the corresponding bit of RTE be set, then the toggle pulse is enabled to the associated high-speed output and that output will toggle upon generation of the toggle pulse from CM2. The structure of these toggled outputs is different from the other high-speed outputs in that the toggling is actually accomplished in a separate toggled flip-flop and not directly in the port latch. This toggle flip-flop cannot be controlled directly by software and powers up in an indeterminate state. The state of the toggle flip-flops is readable in STE bits 6 and 7.

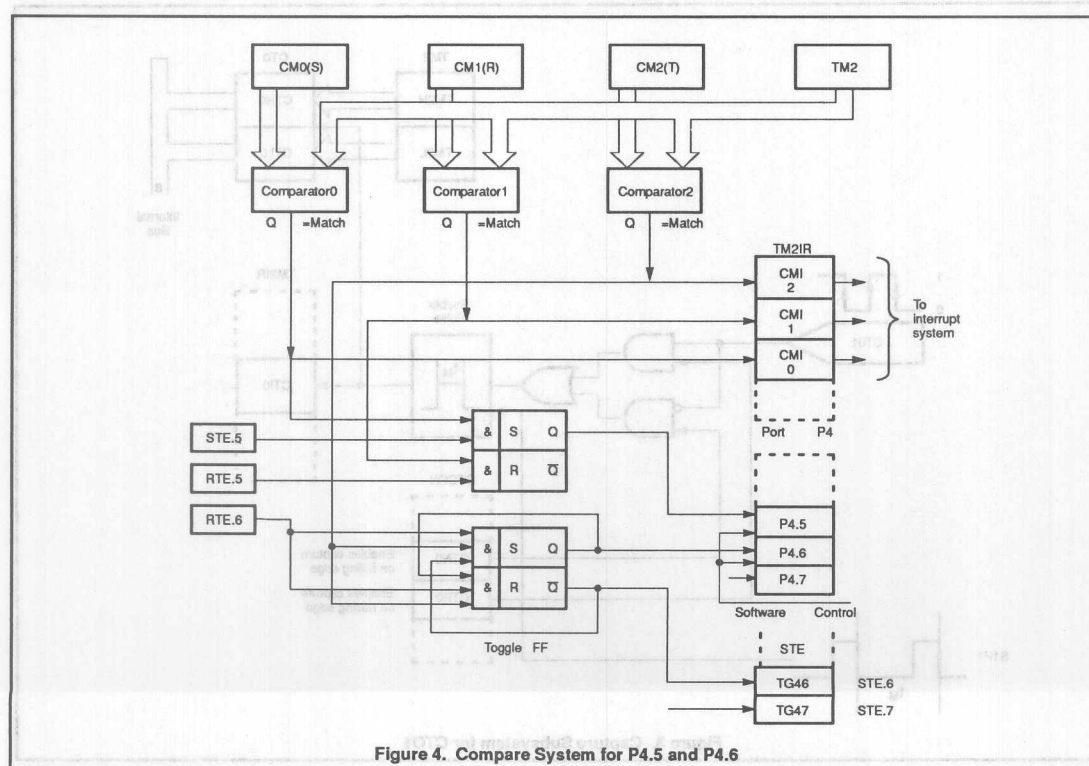


Figure 4. Compare System for P4.5 and P4.6

Counter/timer 2 of the 83C552 microcontroller AN418

APPLICATION EXAMPLE—TIMED FUEL INJECTION

In modern automobiles, optimal combustion is necessary to meet emission standards and improve fuel consumption. Optimal combustion depends on several factors and is enhanced by proper fuel injection based upon these factors which vary according to engine speed and other factors. Thus the task is to control the opening and closing of the engine fuel injectors of each cylinder relative to the crankshaft reference point.

For the application example here, we will not consider the factors which determine the timing relationships. These are assumed to be given quantities. The example here will focus upon the implementation of the injector timing control signals and how they are generated using the Counter/Timer 2 system. The illustration considers a four cylinder engine. While this is an automotive application which serves to clearly illustrate Counter/Timer 2 Subsystem operation, it is clear that many systems share similar timing requirements, and the techniques employed here are applicable to a wide class of timing tasks. The 83C552 will also support six cylinder engine control.

Figure 5 shows the injection timing required for two consecutive revolutions of the engine crankshaft. Start and stop of the injection are given relative to a reference point on the crankshaft. The cylinders are numbered in the order of the injection sequence (not with reference to their physical location). Start of the injection is usually given in angular measure with respect to top dead center, and

the injection duration is assumed to be a time value calculated from engine environmental factors and operating parameters. The angle for the start of the injection must be converted into time with respect to the reference point.

The injector drivers are assumed to be connected to the port 4 high-speed outputs CMSR0 through CMSR3. To obtain the top dead center reference point, the signal from the appropriate sensor is connected to the capture input CT0I. The interrupt for this capture input is enabled so that software can synchronize its operation to this time reference and make use of the top dead center time in the injector timing calculations. The software synchronization takes two forms. First, the captured time is an absolute reference for all real-time output operations. This time is available in capture register CT0. Note that at 12 MHz operation, Timer 2 can have a resolution as fine as 1 microsecond with a total time before overflow of over 65 milliseconds, and these times are adjustable by increasing the prescaler divide ratio. A proper selection can make the timing calculations relatively simple. Second, at the time the input is captured, flags which keep track of the phases of the crankshaft cycle are reset when cylinder 1 is at top dead center. These flags are used in the interrupt service routines to tell which action is required for that phase of the crankshaft.

Consider now the sequence of events in one rotation of the engine crankshaft and refer to Figure 5 during the discussion. Assume that the engine is running, that all relevant

parameters are available, and that it has been determined that the processor is responding to the interrupt associated with the top dead center capture, CT0I. Interrupts for CT0I, CM0 (compare register 0), and CM1 (compare register 1) are enabled. Upon entering the interrupt service routine for CT0I, the previous value of the captured top dead center time is subtracted from the present value, and the crankshaft rotation time is determined. This is used to compute the time to open the first injector from the required angle at which the injector is to open. This time is made available for the interrupt service routine which responds to a compare from CM0. The interrupt service routine is exited.

The next interrupt to occur per the figure for this example is a result of a compare with CM1 which will be a result of the injector stop time for cylinder 4 having been reached. The flags in an internal status register are employed to keep track of the cylinder number that is presently active for both injection stop and injection start times. After identifying this interrupt from the flags, the processor uses the injector start time for cylinder 1 (previously loaded into CM0) and the predetermined duration to calculate the injector stop time for cylinder 1. This value is loaded into compare register CM1 and the reset enable bit for high-speed output CMSR0 is programmed to a 1. This is bit RTE.0 The reset enable bit for the cylinder 4 injector is set to 0 (bit RET.3). The interrupt routine is now exited.

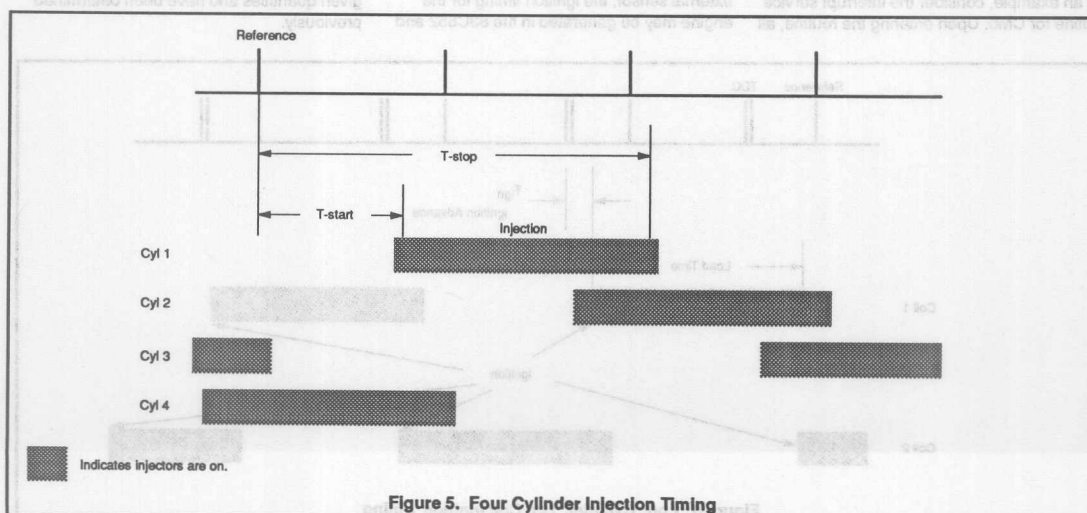


Figure 5. Four Cylinder Injection Timing

Counter/timer 2 of the 83C552 microcontroller AN418

The next interrupt to occur will be for the start time for the injector for cylinder 2. This and all subsequent cases follow the same sequence of events as for the cylinder 1 CM0 interrupt described above. In this case, calculations are made for cylinder 2 and loaded into CM0.STE.1 is programmed to a 1, and STE.0 is programmed to a 0. Similarly, the next interrupt for CM1 is treated in the same way, and the sequence of events rotates around through all cylinders in turn. The flag bits associated with this operation keep track of the injector sequencing.

While this example shows the injection stop time of one cylinder overlapping into the injector on time of the subsequent cylinder, close examination of the operations described above reveal that the start and stop events are independent and can overlap or not as required. In this way all injectors may be driven independently and have overlapping on times.

Given that this is an example applicable to general usage, it is possible that interrupt service routine could be relatively long as it would be in an actual injector application. Since the service routine has other interrupts disabled, the length may cause real-time conflicts. To eliminate this potential problem, the interrupt service routines are divided into two parts. In the first part, all other interrupts are disabled, and the essential register loading is done to prepare for the next interrupt. After this is completed, all interrupts are enabled and the ancillary service routine functions are performed prior to a return to the main routine.

As an example, consider the interrupt service routine for CM0. Upon entering the routine, all

interrupts are disabled. Then the following actions are performed:

- Set bit in STE to start next injector
- Clear bit in STE for injector just started
- Load CM0 with start time for next injector
- Clear CMIO interrupt flag in TM2IR

Now that the essential set-up is made for the next interrupt, all interrupts are now enabled. However, the return to the main program is not invoked until the following ancillary processing is completed:

- Calculate the next absolute start time for the next injector (the next load value for CM0)
- Increment the flag so that the next entry to this interrupt service routine will be able to identify the next injector to start.

The process performing these calculations can be interrupted to service real-time functions.

APPLICATION EXAMPLE—TIMED IGNITION

In electronic ignition systems, multiple ignition coils may be used and each coil is fired by electronic means rather than with the old style mechanical breakers. In a four cylinder engine, there may be two ignition coils, one coil providing spark for a pair of cylinders. Both plugs fire at the same time. For one cylinder, the spark occurs at the appropriate time while for the other cylinder, the spark occurs at the end of the exhaust stroke and has no effect. With timing references to crankshaft top dead center provided by an external sensor, the ignition timing for the engine may be generated in the 83C552 and

applied to the electronic drivers for the ignition coils.

To illustrate the toggle high-speed outputs of the 83C552 Counter/Timer 2 subsystem, the following example will discuss the ignition timing in a four cylinder engine employing the two coil approach with one coil for a pair of cylinders. The coil timing is illustrated in Figure 6. A reference time is used which is a given interval prior to top dead center so that the times used in the illustration can be always after the reference. There are two times of interest for each coil: the load time and the ignition point.

Ignition advance is usually given in degrees crankshaft angle prior to top dead center. As with injection, this angle is assumed to be derived from other calculations and is a given value for this illustration. This angle must be converted in to a time with respect to the reference point. The load time (the time at which the coil has to be switched on to reach the current that will give sufficient energy for an adequate spark) must be subtracted from the desired ignition point. At the ignition time, the coil will be switched off and the spark will be generated.

The coil driver electronics are connected to port bits P4.6 and P4.7. Ignition coil 1 is connected to P4.6, and ignition coil 2 is connected to P4.7. These outputs are the toggle high-speed outputs controlled by the 16-bit compare register, CM2. The program simply needs to set up the compare and control registers to turn the coils on and off at the appropriate times. It is assumed in this example that the ignition and load times are given quantities and have been determined previously.

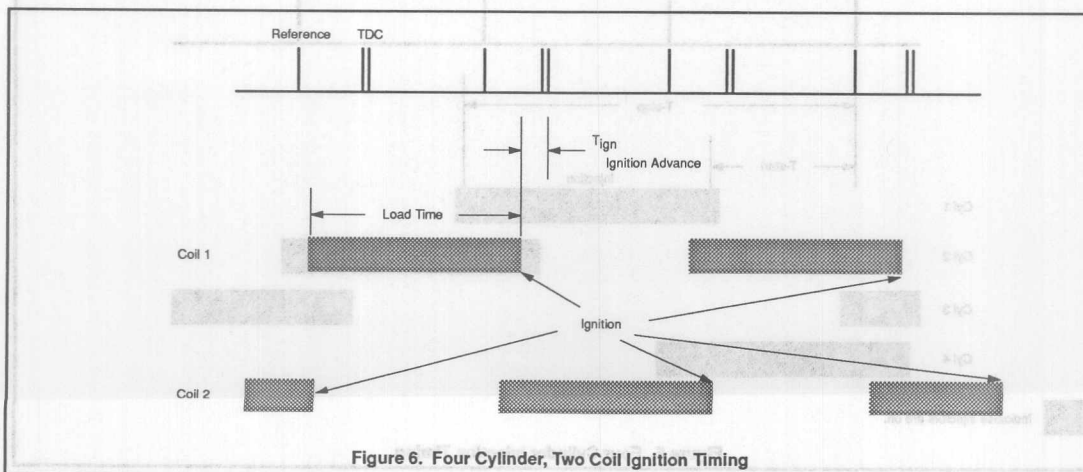


Figure 6. Four Cylinder, Two Coil Ignition Timing

Counter/timer 2 of the 83C552 microcontroller

AN418

Consider now the sequence of events in two rotations of the engine crankshaft and refer to Figure 6. Assume that the engine is running, that all relevant parameters are available, and that it has been determined that the processor is responding to the interrupt associated with a compare to CM2. The top dead center time and crankshaft rotation speed have been already determined through the top dead center capture, CTOI. This is the same as in the injector example. The interrupt for CTOI is enabled. From the top dead center time, the times to turn on and turn off the coil drivers are computed and made available in data storage locations in the microcontroller. It is also convenient to have flags to identify the step in the complete ignition cycle. The flags are cleared in the interrupt service routine for top dead center of cylinder 1.

Upon entering the interrupt service routine, other interrupts are disabled. Examination of the flags reveals that the state of the ignition sequence is that coil 1 has been turned on to begin the current build up (load time). The next event will therefore be turning off coil 2 to cause ignition. The interrupt service routine then performs the following actions: The time to turn off coil 2 is moved into compare register 2, CM2. Bit 6 of RTE is cleared; this disconnects the output of CM2 from the toggle flip-flop of P4.6 (coil 1). Bit 7 of RTE is set; this connects the output of CM2 to the toggle flip-flop of P4.7 (coil 2). The flags are

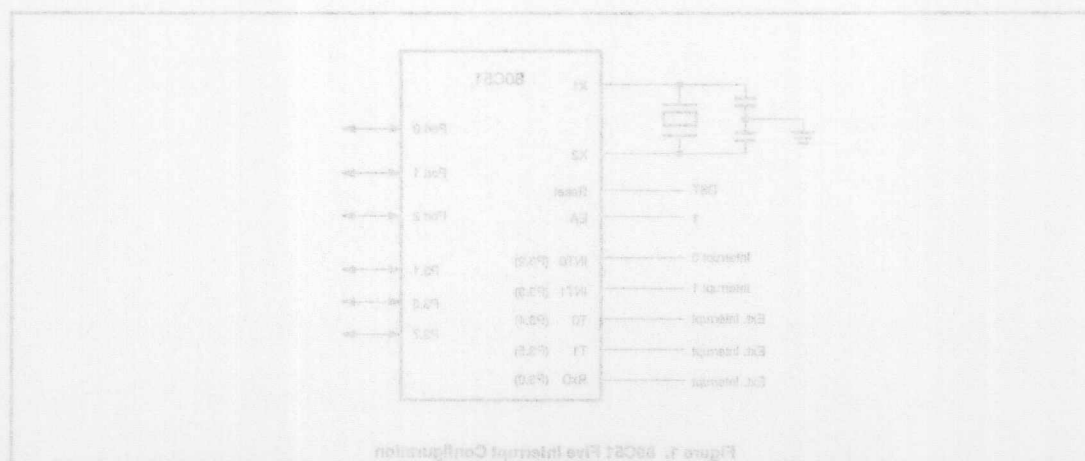
incremented to indicate that the next interrupt will be a result of coil 2 turning off and causing ignition. The other interrupts can be enabled and a return to the main program can be executed. After the other interrupts are enabled and before a return is made to the main program, it may be convenient to do any necessary calculations to determine the time value to be loaded into CM2 in the next CM2 interrupt.

Since the flip-flops are toggled, it is likely that upon power up of the microcontroller, the toggle flip-flops will not be in the desired state. To get the toggle flip-flops in the correct state in the ignition cycle, the flip-flops must be toggled if they are in the wrong state. To determine if this is necessary, the state of the toggle flip-flops can be read from the STE register. The state of the P4.6 flip-flop is present in STE bit 6 and the state of the P4.7 flip-flop is present in STE bit 7. Comparing the actual state to the required state determines which if any or both of the flip-flops must be toggled. If a toggle is necessary to put one or both of the flip-flops in the correct state, the corresponding bits in RTE would be set for those flip-flops requiring the toggle, and CM2 would be loaded with a value that is slightly larger than the present contents of timer 2. If desired for reliability purposes, the state of the flip-flops could be checked periodically against the ignition cycle flags to determine if a correction is necessary.

CONCLUSION

This application note has examined one aspect of the 83C552 CMOS 80C51 derivative microcontroller. The Counter/Timer 2 Subsystem has been applied to a complex timing task of gasoline engine injector valve and ignition coil timing control. While this is a specific application to the automotive interests, the results are applicable to a wide variety of time measurement and control applications. The 83C552 would be ideal for many electromechanical systems such as copy machines, fax machines, industrial process control equipment, automatic transmission control, and anti-skid and anti-lock braking control.

These application areas are those which can successfully employ the 83C552 Counter/Timer 2; however, the other features should not be overlooked. When combined with the 10-bit A to D Converter, the Pulse Width Modulator, the I²C serial bus, and peripheral device family, the 83C552 provides minimum component count solutions for cellular radio systems, professional audio systems, and medical instrumentation products such as bedside patient monitors and analyzers for home care and sports use.



Using up to 5 external interrupts on 80C51 family microcontrollers

AN420

80C51 family microcontrollers are equipped with up to two inputs which may be used as general-purpose interrupts. A typical device provides a total of 5 interrupt sources. Timer 0 and Timer 1 generate vectored interrupts, as does the Serial Port. Applications that require more than two externally signaled vectored interrupts, and do not use one or more of the counters or the serial port, can be configured to use these facilities for additional external interrupt inputs.

This note describes a method to configure the timer/counters and the serial port for use as interrupt inputs (see Figure 1). Minimum response time is a goal for this configuration.

Another popular method to implement extra interrupt inputs is to poll under software control a port pin configured as an input. This method is necessary when the on-chip peripherals are in use. Applications where this approach is recommended are ones in which the processor spends more than half of the time executing a "wait loop," or a short code sequence which jumps or branches back on itself without performing any functions. In this case, the instructions that will check the state of input used as an interrupt source are inserted into this sequence. Consequently, this input is ignored when other routines are being executed. This input may have to be latched externally, or the processor may miss the signal while executing other routines.

Dedicated interrupt inputs that vector the processor to individual service routines (as the two general-purpose interrupt inputs work) do not have the drawbacks of the method described above.

COUNTER/TIMER CONFIGURATION

Timers 0 and 1 are placed in mode 2, which configures the timer/register as an 8-bit counter with automatic reload. The counter and reload register are loaded with FF hexadecimal which is stored in TH1 and TL1 or TH0 and TL0.

To prepare one of the timers for this kind of operation, a number of control bits have to be set up. The following is a list of these bits and their values:

In TMOD: In TCON: In IE:
GATE = 0 TRi = 1 ETi = 1
C/T = 1
M1 = 1 EA = 1
MO = 0

Where "i" is the timer number being used as the external interrupt. The TMOD value would be 66 hexadecimal if both timers are being used as external interrupt sources, x6 hex for timer 0, and 6x hex for timer 1. The interrupt priority may also be set in the IP register.

A falling edge on the corresponding Timer 0 or Timer 1 input (T0 or T1) will cause the

counter to overflow and generate a timer interrupt. The counter will be automatically loaded with another FF from the reload register, so the interrupt can occur again as soon as the interrupt service routine completes. Counter/Timer operation is described in detail elsewhere in this manual.

SERIAL PORT CONFIGURATION

The serial port can be placed in mode 2, which is a 9-bit UART with the baud rate derived from the oscillator. The external interrupt is signaled through this port on the RxD receive data pin. Reception is initiated by a detected 1-to-0 transition at RxD. The signal must stay at 0 for at least five-eighths of a bit period for this level to be recognized. Refer to the description of baud rates to determine the length of a bit period at the oscillator frequency selected for the application. The input signal should remain low for at least one bit period and for not more than 9 bit periods.

To prepare the serial port for use as an external interrupt, the following bits must be set up:

In SCON:
SM0 = 1
SM1 = 0
SM2 = 0
REN = 1

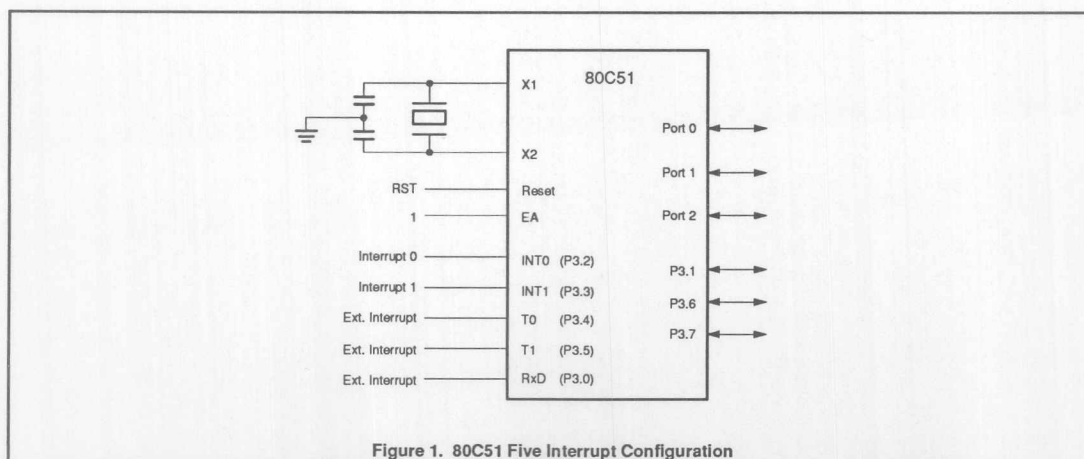


Figure 1. 80C51 Five Interrupt Configuration

Using up to 5 external interrupts on 80C51 family microcontrollers

AN420

The Serial Port Interrupt is then used as a general-purpose interrupt. The contents of receive buffer should be ignored, and will subsequently be overwritten during the next interrupt.

; Demonstration program for five external interrupts.

\$MOD51

\$TITLE (Five Vectors External Interrupts)

; Interrupt Jump Table

```
ORG OH
AJMP Setup
```

```
ORG 3H
RETI
```

```
ORG 0BH
AJMP Tim0
```

```
ORG 13H
RETI
```

```
ORG 1BH
AJMP Tim1
```

```
ORG 23H
AJMP Serial
```

; Begin setup code

```
Setup MOV SP,#7FH
```

; Configure both timers

```
MOV TMOD,#66H
MOV A,#0FFH
MOV TLO,A
MOV TH0,A
MOV TL1,A
MOV TH1,A
SETB ET0
SETB ET1
SETB TR0
SETB TR1
```

; Configure the serial port

```
SETB ES
MOV SCON,#90H
SETB EA
```

```
Wait: NOP
      JMP Wait
```

```
Serial: NOP
        CLR RI
        RETI
```

```
Tim0: NOP
      RETI
```

```
Tim1: NOP
      RETI
```

END

Note that the response time for this input will be slower than for the Counter/Timer inputs. This is due to the fact that the RI is generated after the eighth serial data bit time after the falling edge on Rx/D.

;Reset

;External interrupt 0.
(not implemented in this demo)

;Timer 0 interrupt.

;External interrupt 1.
(not implemented in this demo)

;Timer 1 interrupt.

;Serial port interrupt.

;Initialize the stack pointer.

;Put both counters into mode 2.

;Load FF hex into both counters

;Enable Timer 0 interrupt.

;Enable Timer 1 interrupt.

;Enable Timer 0 to run.

;Enable Timer 1 to run.

;Enable serial port interrupt.

;Put the serial port in mode 2.

;Enable interrupt system.

;Wait for an interrupt.

;Serial interrupt service routine.
;Clear receiver interrupt flag.

;Timer 0 interrupt service routine.

;Timer 0 interrupt service routine.

8051 family warm boot determinations

AN424

DESCRIPTION

For some classes of applications, it may be desirable to know if the application of the reset signal to a microcontroller is due to an initial power-on sequence, or is the result of an external signal such as an operator pressing a reset pushbutton, or the result of a watchdog timer or similar event.

While there are perhaps numerous hardware solutions that can be employed, a simple software solution can offer a high degree of confidence in making this determination. The task is to determine the differences in state of resources internal to the microcontroller that would occur as a result of these two types of reset conditions. With respect to the 80C51 family of microcontrollers, on-chip resources consist of the special function registers (SFRs) and the internal data memory (RAM). Most of the SFR locations are initialized as a result of a reset condition and thus cannot be used for this determination. The data memory contents are unaffected by reset. Thus, valid data loaded into the RAM of the 80C51 while executing a program would not be affected by the application of an external reset signal provided the power source for the microcontroller has not been removed (as is the case for a "warm boot").

The contents of data memory as a result of an initial application of power, however, is indeterminate. While this effect has not been extensively characterized, empirical observation suggests that it is highly random in nature. If it is assumed, for the moment, that the behavior of a given byte of data memory is such that it will power-up with a

value that is totally random, then there is a one in eight chance that it will power-up with a predetermined value. If the assumption is extended to two bytes, a 16-bit number, then there is one in 2^{16} chance that both bytes will power-up with predetermined values. Extending this to four bytes results in a one in 2^{32} chance; a very small probability. This is the basis for the software determination of a warm or cold boot condition.

The technique consists of evaluating the contents of four consecutive bytes of data memory following a reset condition to determine whether these bytes had been previously loaded with known data values. If the contents of all four bytes match predetermined values, this is interpreted to be a warm boot condition. If there is no match, it is then interpreted to be a cold boot condition. At this point, it is necessary to load these four bytes with predetermined data to prepare for the possibility of a subsequent warm boot condition.

The software example included in this application brief can be used to perform this warm or cold boot determination.

The symbols WARM1 through WARM4 represent the predetermined values. The symbol WARM is the address of the first of the four consecutive bytes in data memory. It is set to 30H to avoid conflict with the four register banks, the stack, and the bit-addressable locations in data memory. The symbol WARMBT is a bit-addressable location used as a status bit. It is set as the

result of a warm boot and cleared as a result of a cold boot.

The label START is the location of the first instruction to be executed following a reset (address = 0000H). An instruction is located here to jump into the main body of the program to bypass the interrupt vector locations.

The main program body begins by loading register R0 with the address of the first byte in data memory to be evaluated. The contents of this first byte is compared with the first predetermined value. If there is no match, the conclusion is that it is a cold boot. However, if a match is found, this does not imply that it is a warm boot since all four bytes must match, and therefore the remaining three bytes must also be evaluated. Register R0 is incremented to point to the second byte and then compared to the second predetermined value. Comparison of the bytes proceeds until either a no match condition is found or until all four bytes have been evaluated successfully. If all four bytes compared favorable, then a status bit (WARMBT) is set to indicate a warm boot and the remainder of the application program is completed.

An unsuccessful comparison results in branching to the label COLD. This section of code clears the status bit (WARMBT) to indicate a cold boot, and loads the four bytes of data memory with the predetermined values preparing the system for a subsequent possible warm boot. Program flow then continues with the remainder of the application program.

8051 family warm boot determinations

AN424

1
2 ;warm boot application example
3
4 WARM EQU 30H ;first location of the four bytes in RAM
5 WARM1 EQU 55H ;first predetermined value
6 WARM2 EQU 0AAH ;second predetermined value
7 WARM3 EQU 33H ;third predetermined value
8 WARM4 EQU 0CCH ;fourth predetermined value
9 WARMBT EQU 0 ;warm boot status bit
10
11 ORG 0
12
13 START: JMP MAIN ;bypass interrupt vectors
14
15 ORG 26H
16
17 MAIN: MOV R0, #WARM ;pointer for first byte
18 CJNE @R0, #WARM1, COLD ;test first byte
19 INC R0 ;pointer for second byte
20 CJNE @R0, #WARM2, COLD ;test second byte
21 INC R0 ;pointer for third byte
22 CJNE @R0, #WARM3, COLD ;test third byte
23 INC R0 ;pointer for fourth byte
24 CJNE @R0, #WARM4, COLD ;test fourth byte
25 SETB WARMBT ;this is a warm start
26 JMP INIT ;continue with rest of application
27 COLD: CLR WARMBT ;this is a cold boot
28 MOV R0, #WARM ;pointer for first byte
29 MOV @R0, #WARM1 ;load the four bytes for future test
30 INC R0 ;
31 MOV @R0, #WARM2 ;
32 INC R0 ;
33 MOV @R0, #WARM3 ;
34 INC R0 ;
35 MOV @R0, #WARM4 ;
36 INIT: ;continue with the application
37
38 END

ASSEMBLY COMPLETE, 0 ERRORS FOUND

COLD	C ADDR	003CH	
INIT	C ADDR	004BH	
MAIN	C ADDR	0026H	
START.	C ADDR	0000H	NOT USED
WARM	NUMB	0030H	
WARM1.	NUMB	0055H	
WARM2.	NUMB	00AAH	
WARM3.	NUMB	0033H	
WARM4.	NUMB	00CCH	
WARMBT	NUMB	0000H	

The following program allows an 80C51 family microcontroller to load most of its code into a RAM over a serial link after power up and execute out of the RAM for normal operation. This can allow a final product to have firmware updates done by a simple diskette mailing. Such a program is often called a "bootstrap loader".

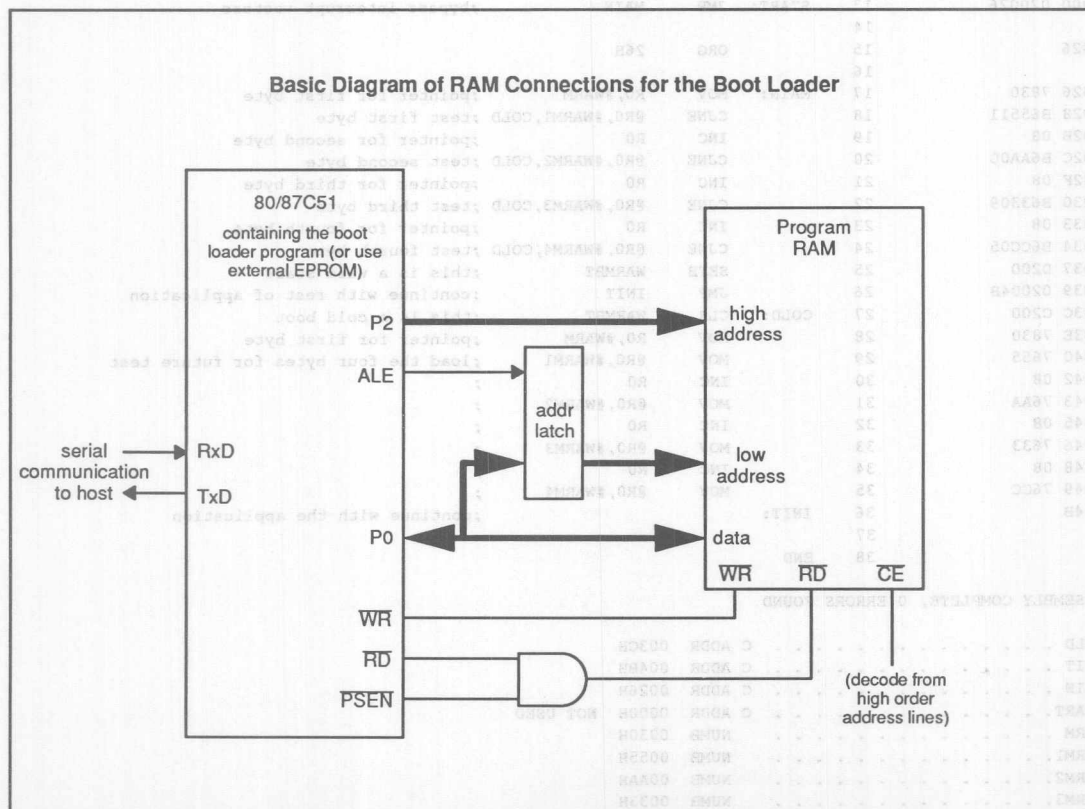
For this example, it is assumed that the code download is done via a serial communication

link, although the program could be adapted to other forms of download. The comments at the beginning of the listing are intended to document the program and its use completely.

An additional comment would be that any static routines (low level routines that are unlikely to change over time) can probably be put into the permanent program memory (on-chip or off-chip ROM or EPROM) along

with the bootstrap loader to save program RAM space for other things.

The source code file for this program is available for downloading from the Philips computer bulletin board system. This system is open to all callers, operates 24 hours a day, and can be accessed with modems at 2400, 1200, and 300 baud. The telephone numbers for the BBS are: (800) 451-6644 (in the U.S. only) or (408) 991-2406.



RAM loader program for 80C51 family applications AN440

```

;=====
;      ("")
;      correct starting address. The format for this is as shown below.
;      followed by the address in ASCII hexadecimal.
;      Example: "00000000"
;      written by G. Goodhue, Philips Electronics

; This program allows downloading a hexadecimal program file over an
; asynchronous serial link to a code RAM in an 80C51 system. The downloaded
; code may then be executed as the main program for the system. This technique
; may be used in a system that normally connects to a host PC so that the code
; may come from a disk and thus be easily updated. The system RAM must be
; wired to the 80C51 system so that it appears as both data and program memory
; (wire the RAM normally, but use the logical AND of RD and PSEN for the
; output enable.)

; To use the bootstrap program, an Intel Hex file is sent through the serial
; port in 8-N-1 format at 9600 baud. The baud rate and format may be altered
; by making small changes in the serial port setup routine (SerStart).

; Note that there is no hardware handshaking (e.g. RTS/CTS or XON/XOFF)
; implemented between the host and the bootstrap system. This was done to keep
; the protocol between the two systems as simple as possible.

; Since the bootstrap program does not echo the data file, there is no chance
; of an overrun unless the 80C51 is running very slowly and/or the
; communication is very fast. An 80C51 running at 11.0592 MHz (the most
; commonly used frequency in systems with serial communication) will be able
; to easily keep up with 38.4K baud communication without handshaking.
;=====

; The download protocol for this program is as follows:

; - When the bootstrap program starts up, it sends a prompt character ("=")
;   up the serial link to the host.

; - The host may then send the hexadecimal program file down the serial link.
;   At any time, the host may send an escape character (1B hex) to abort and
;   restart the download process from scratch, beginning from the "=" prompt.
;   This procedure may be used to restart if a download error occurs.

; - At the end of a hex file download, a colon (":") prompt is returned. If
;   an error or other suspicious circumstance occurred, a flag value will
;   also be returned as shown below. The flag is a bit map of possible
;   conditions and so may represent more than one problem. If an error
;   occurs, the bootstrap program will refuse to execute the downloaded
;   program.

; Exception codes:
;   01 - non-hexadecimal characters found embedded in a data line.
;   02 - bad record type found.
;   04 - incorrect line checksum found.
;   08 - no data found.
;   10 - incremented address overflowed back to zero.
;   20 - RAM data write did not verify correctly.

; - If a download error occurs, the download may be retried by first sending
;   an escape character. Until the escape is received, the bootstrap program
;   will refuse to accept any data and will echo a question mark ("??") for
;   any character sent.

; - After a valid file download, the bootstrap program will send a message
;   containing the file checksum. This is the arithmetic sum of all of the
;   DATA bytes (not addresses, record types, etc.) in the file, truncated to
;   16 bits. This checksum appears in parentheses: "(abcd)". Program

```


RAM loader program for 80C51 family applications AN440

```

; execution may then be started by telling the bootstrap program the
; correct starting address. The format for this is to send a slash ("/")
; followed by the address in ASCII hexadecimal, followed by a carriage
; return. Example: "/8A31<CR>"

; - If the address is accepted, an at sign ("@" ) is returned before executing
; the jump to the downloaded file.

; The bootstrap loader can be configured to re-map interrupt vectors to the
; downloaded program if jumps to the correct addresses are set up. For
; instance, if the program RAM in the system where this program is to be used
; starts at 8000 hexadecimal, the re-mapped interrupts may begin at 8003 for
; external interrupt 0, etc.

;=====
$Title(Bootstrap Loader for Hexadecimal Files)
$Date(04-13-92)
$MOD51

;=====
;
; Definitions
;
;=====

LF      EQU      0Ah      ; Line Feed character.
CR      EQU      0Dh      ; Carriage Return character.
ESC      EQU      1Bh      ; Escape character.
StartChar EQU      ':'      ; Line start character for hex file.
Slash    EQU      '/'      ; Go command character.
Skip     EQU      13      ; Value for "Skip" state.

Ch       DATA     0Fh      ; Last character received.
State    DATA     10h      ; Identifies the state in process.
DataByte DATA     11h      ; Last data byte received.
ByteCount DATA     12h      ; Data byte count from current line.
HighAddr DATA     13h      ; High and low address bytes from the
LowAddr  DATA     14h      ; current data line.
RecType  DATA     15h      ; Line record type for this line.
ChkSum   DATA     16h      ; Calculated checksum received.
HASave  DATA     17h      ; Saves the high and low address bytes
LASave  DATA     18h      ; from the last data line.
FilChkHi DATA     19h      ; File checksum high byte.
FilChkLo DATA     1Ah      ; File checksum low byte.

Flags    DATA     20h      ; State condition flags.
HexFlag  BIT        Flags.0 ; Hex character found.
EndFlag  BIT        Flags.1 ; End record found.
DoneFlag BIT        Flags.2 ; Processing done (end record or some
; kind of error.

EFlags   DATA     21h      ; Exception flags.
ErrFlag1 BIT        EFlags.0 ; Non-hex character embedded in data.
ErrFlag2 BIT        EFlags.1 ; Bad record type.
ErrFlag3 BIT        EFlags.2 ; Bad line checksum.
ErrFlag4 BIT        EFlags.3 ; No data found.
ErrFlag5 BIT        EFlags.4 ; Incremented address overflow.
ErrFlag6 BIT        EFlags.5 ; Data storage verify error.
DatSkipFlag BIT      Flags.3 ; Any data found should be ignored.

```


RAM loader program for 80C51 family applications AN440

```

;=====
;                               Reset and Interrupt Vectors
;=====
; The following are dummy labels for re-mapped interrupt vectors. The
; addresses should be changed to match the memory map of the target system.

ExInt0    EQU    8003h    ; Remap address for ext interrupt 0.
T0Int     EQU    800Bh    ; Timer 0 interrupt.
ExInt1    EQU    8013h    ; External interrupt 1.
T1Int     EQU    801Bh    ; Timer 1 interrupt.
SerInt     EQU    8023h    ; Serial port interrupt.

                ORG    0000h
                LJMP   Start    ; Go to the downloader program.

; The following are intended to allow re-mapping the interrupt vectors to the
; users downloaded program. The jump addresses should be adjusted to reflect
; the memory mapping used in the actual application.

; Other (or different) interrupt vectors may need to be added if the target
; processor is not an 80C51.

                ORG    0003h
                LJMP   ExInt0    ; External interrupt 0.
                RETI

                ORG    000Bh
                LJMP   T0Int     ; Timer 0 interrupt.
                RETI

                ORG    0013h
                LJMP   ExInt1    ; External interrupt 1.
                RETI

                ORG    001Bh
                LJMP   T1Int     ; Timer 1 interrupt.
                RETI

                ORG    0023h
                LJMP   SerInt    ; Serial port interrupt.
                RETI

;=====
;                               Reset and Interrupt Vectors
;=====
Start:        MOV    IE,#0    ; Turn off all interrupts.
                MOV    SP,#5Fh    ; Start stack near top of '51 RAM.
                ACALL  SerStart    ; Setup and start serial port.
                ACALL  CRLF    ; Send a prompt that we are here.
                MOV    A,#'='    ; "<CRLF> ="
                ACALL  PutChar    ;
                ACALL  HexIn    ; Try to read hex file from serial port.

                ACALL  ErrPrt    ; Send a message for any errors or
                                ; warnings that were noted.
                MOV    A,EFlags    ; We want to get stuck if a fatal
                JZ     HexOK    ; error occurred.

ErrLoop:      MOV    A,#'?'    ; Send a prompt to confirm that we

```

RAM loader program for 80C51 family applications AN440

```

ACALL PutChar      ; are 'stuck'. " ? "
ACALL GetChar      ; Wait for escape char to flag reload.
SJMP ErrLoop

HexOK: MOV EFlags,#0 ; Clear errors flag in case we re-try.
ACALL GetChar      ; Look for GO command.
CJNE A,#Slash,HexOK ; Ignore other characters received.

ACALL GetByte      ; Get the GO high address byte.
JB ErrFlag1,HexOK  ; If non-hex char found, try again.
MOV HighAddr,DataByte ; Save upper GO address byte.

ACALL GetByte      ; Get the GO low address byte.
JB ErrFlag1,HexOK  ; If non-hex char found, try again.
MOV LowAddr,DataByte ; Save the lower GO address byte.

ACALL GetChar      ; Look for CR.
CJNE A,#CR,HexOK   ; Re-try if CR not there.

; All conditions are met, so hope the data file and the GO address are all
; correct, because now we're committed.

MOV A,#'@'        ; Send confirmation to GO. "@ "
ACALL PutChar
JNB TI,$          ; Wait for completion before GOing.
PUSH LowAddr      ; Put the GO address on the stack,
PUSH HighAddr     ; so we can Return to it.
RET              ; Finally, go execute the user program!

;=====
; Hexadecimal File Input Routine
;=====

HexIn: CLR A      ; Clear out some variables.
MOV State,A
MOV Flags,A
MOV HighAddr,A
MOV LowAddr,A
MOV HASave,A
MOV LASave,A
MOV ChkSum,A
MOV FilChkHi,A
MOV FilChkLo,A
MOV EFlags,A
SETB ErrFlag4 ; Start with a 'no data' condition.

StateLoop: ACALL GetChar ; Get a character for processing.
ACALL AscHex ; Convert ASCII-hex character to hex.
MOV Ch,A ; Save result for later.
ACALL GoState ; Go find the next state based on
; this char.
JNB DoneFlag,StateLoop ; Repeat until done or terminated.

ACALL PutChar ; Send the file checksum back as
MOV A,#'(' ; confirmation. " (abcd) "
ACALL PutChar
MOV A,FilChkHi
ACALL PrByte
MOV A,FilChkLo
ACALL PrByte
MOV A,#')'
ACALL PutChar
ACALL CRLF

```

RAM loader program for 80C51 family applications AN440

```

RET                                ; Exit to main program.

; Find and execute the state routine pointed to by "State".

GoState:  MOV     A,State           ; Get current state.
          ANL     A,#0Fh           ; Insure branch is within table range.
          RL      A                ; Adjust offset for 2 byte insts.
          MOV     DPTR,#StateTable ; Go to appropriate state.
          JMP     @A+DPTR

StateTable: AJMP   StWait          ; 0 - Wait for start.
            AJMP   StLeft          ; 1 - First nibble of count.
            AJMP   StGetCnt        ; 2 - Get count.
            AJMP   StLeft          ; 3 - First nibble of address byte 1.
            AJMP   StGetAd1        ; 4 - Get address byte 1.
            AJMP   StLeft          ; 5 - First nibble of address byte 2.
            AJMP   StGetAd2        ; 6 - Get address byte 2.
            AJMP   StLeft          ; 7 - First nibble of record type.
            AJMP   StGetRec        ; 8 - Get record type.
            AJMP   StLeft          ; 9 - First nibble of data byte.
            AJMP   StGetDat        ; 10 - Get data byte.
            AJMP   StLeft          ; 11 - First nibble of checksum.
            AJMP   StGetChk        ; 12 - Get checksum.
            AJMP   StSkip          ; 13 - Skip data after error condition.
            AJMP   BadState        ; 14 - Should never get here.
            AJMP   BadState        ; 15 - " " " " " "

; This state is used to wait for a line start character. Any other characters
; received prior to the line start are simply ignored.

StWait:   MOV     A,Ch             ; Retrieve input character.
          CJNE    A,#StartChar,SWEX ; Check for line start.
          INC     State            ; Received line start.
SWEX:     RET

; Process the first nibble of any hex byte.

StLeft:   MOV     A,Ch             ; Retrieve input character.
          JNB     HexFlag,SLERR    ; Check for hex character.
          ANL     A,#0Fh           ; Isolate one nibble.
          SWAP    A                ; Move nibble too upper location.
          MOV     DataByte,A       ; Save left/upper nibble.
          INC     State            ; Go to next state.
          RET      ; Return to state loop.

SLERR:    SETB    ErrFlag1         ; Error - non-hex character found.
          SETB    DoneFlag         ; File considered corrupt. Tell main.
          RET

; Process the second nibble of any hex byte.

StRight:  MOV     A,Ch             ; Retrieve input character.
          JNB     HexFlag,SRERR    ; Check for hex character.
          ANL     A,#0Fh           ; Isolate one nibble.
          ORL     A,DataByte       ; Complete one byte.
          MOV     DataByte,A       ; Save data byte.
          ADD     A,ChkSum         ; Update line checksum,
          MOV     ChkSum,A         ; and save.
          RET      ; Return to state loop.

SRERR:    SETB    ErrFlag1         ; Error - non-hex character found.

```

RAM loader program for 80C51 family applications AN440

```

SETB    DoneFlag        ; File considered corrupt. Tell main.
RET

; Find and execute the state routine pointed to by "State".
; Get data byte count for line.
StGetCnt: ACALL    StRight    ; Complete the data count byte.
          MOV      A,DataByte
          MOV      ByteCount,A
          INC      State      ; Go to next state.
          RET              ; Return to state loop.

; Get upper address byte for line.
StGetAd1: ACALL    StRight    ; Complete the upper address byte.
          MOV      A,DataByte
          MOV      HighAddr,A ; Save new high address.
          INC      State      ; Go to next state.
          RET              ; Return to state loop.

; Get lower address byte for line.
StGetAd2: ACALL    StRight    ; Complete the lower address byte.
          MOV      A,DataByte
          MOV      LowAddr,A  ; Save new low address.
          INC      State      ; Go to next state.
          RET              ; Return to state loop.

; Get record type for line.
StGetRec: ACALL    StRight    ; Complete the record type byte.
          MOV      A,DataByte
          MOV      RecType,A
          JZ        SGRDat    ; This is a data record.
          CJNE     A,#1,SGRErr ; Check for end record.
          SETB     EndFlag    ; This is an end record.
          SETB     DatSkipFlag ; Ignore data embedded in end record.
          MOV      State,#11  ; Go to checksum for end record.
          SJMP     SGREX

SGRDat:    INC      State      ; Go to next state.
SGREX:     RET              ; Return to state loop.

SGRErr:    SETB     ErrFlag2   ; Error, bad record type.
          SETB     DoneFlag    ; File considered corrupt. Tell main.
          RET

; Get a data byte.
StGetDat: ACALL    StRight    ; Complete the data byte.
          JB       DatSkipFlag,SGD1 ; Don't process the data if the skip
                                     ; flag is on.
          ACALL    Store      ; Store data byte in memory.
          MOV      A,DataByte
          ADD      A,FilChkLo  ; Update the file checksum,
          MOV      FilChkLo,A ; which is a two-byte summation of
          CLR      A           ; all data bytes.
          ADDC     A,FilChkHi
          MOV      FilChkHi,A
          MOV      A,DataByte

```

RAM loader program for 80C51 family applications AN440

```

SGD1:      DJNZ      ByteCount,SGDEX      ; Last data byte?
           INC       State                ; Done with data, go to next state.
           SJMP      SGDEX2

SGDEX:     DEC       State                ; Set up state for next data byte.
SGDEX2:    RET                          ; Return to state loop.

; Get checksum.

StGetChk:  ACALL     StRight              ; Complete the checksum byte.
           JNB       EndFlag,SGC1        ; Check for an end record.
           SETB      DoneFlag            ; If this was an end record,
           SJMP      SGCEX              ; we are done.

SGC1:      MOV       A,ChkSum             ; Get calculated checksum.
           JNZ       SGCErr              ; Result should be zero.
           MOV       ChkSum,#0           ; Preset checksum for next line.
           MOV       State,#0            ; Line done, go back to wait state.
           MOV       LASave,LowAddr      ; Save address byte from this line for
           MOV       HASave,HighAddr    ; later check.
SGCEX:     RET                          ; Return to state loop.

SGCErr:    SETB      ErrFlag3             ; Line checksum error.
           SETB      DoneFlag            ; File considered corrupt. Tell main.
           RET

; This state used to skip through any additional data sent, ignoring it.

StSkip:    RET                          ; Return to state loop.

; A place to go if an illegal state comes up somehow.

BadState:  MOV       State,#Skip          ; If we get here, something very bad
           RET                          ; happened, so return to state loop.

; Store - Save data byte in external RAM at specified address.

Store:     MOV       DPH,HighAddr        ; Set up external RAM address in DPTR.
           MOV       DPL,LowAddr
           MOV       A,DataByte
           MOVX      @DPTR,A             ; Store the data.

           MOVX      A,@DPTR             ; Read back data for integrity check.
           CJNE      A,DataByte,StoreErr ; Is read back OK?

           CLR       ErrFlag4            ; Show that we've found some data.
           INC       DPTR               ; Advance to the next addr in sequence.
           MOV       HighAddr,DPH       ; Save the new address
           MOV       LowAddr,DPL
           CLR       A
           CJNE      A,HighAddr,StoreEx ; Check for address overflow
           CJNE      A,LowAddr,StoreEx ; (both bytes are 0).
           SETB      ErrFlag5            ; Set warning for address overflow.
StoreEx:    RET

StoreErr:  SETB      ErrFlag6            ; Data storage verify error.
           SETB      DoneFlag            ; File considered corrupt. Tell main.
           RET

```

RAM loader program for 80C51 family applications AN440

```

;----- Subroutines -----
;-----
; Subroutine summary:
; SerStart - Serial port setup and start.
; GetChar - Get a character from the serial port for processing.
; GetByte - Get a hex byte from the serial port for processing.
; PutChar - Output a character to the serial port.
; AscHex - See if char in ACC is ASCII-hex and if so convert to hex nibble.
; HexAsc - Convert a hexadecimal nibble to its ASCII character equivalent.
; ErrPrt - Return any error codes to our host.
; CRLF - output a carriage return / line feed pair to the serial port.
; PrByte - Send a byte out the serial port in ASCII hexadecimal format.

; SerStart - Serial port setup and start.
SerStart: MOV A,PCON ; Make sure SMOD is off.
CLR ACC.7
MOV PCON,A
MOV TH1,#0FDh ; Set up timer 1.
MOV TLO,#0FDh
MOV TMOD,#20h
MOV TCON,#40h
MOV SCON,#52h ; Set up serial port.
RET

; GetByte - Get a hex byte from the serial port for processing.
GetByte: ACALL GetChar ; Get first character of byte.
ACALL AscHex ; Convert to hex.
MOV Ch,A ; Save result for later.
ACALL StLeft ; Process as top nibble of a hex byte.
ACALL GetChar ; Get second character of byte.
ACALL AscHex ; Convert to hex.
MOV Ch,A ; Save result for later.
ACALL StRight ; Process as bottom nibble of hex byte.
RET

; GetChar - Get a character from the serial port for processing.
GetChar: JNB RI,$ ; Wait for receiver flag.
CLR RI ; Clear receiver flag.
MOV A,SBUF ; Read character.
CJNE A,#ESC,GCEX ; Re-start immediately if Escape char.
LJMP Start
GCEX: RET

; PutChar - Output a character to the serial port.
PutChar: JNB TI,$ ; Wait for transmitter flag.
CLR TI ; Clear transmitter flag.
MOV SBUF,A ; Send character.
RET

; AscHex - See if char in ACC is ASCII-hex and if so convert to a hex nibble.
; Returns nibble in A, HexFlag tells if char was really hex. The ACC is not
; altered if the character is not ASCII hex. Upper and lower case letters
; are recognized.

```


RAM loader program for 80C51 family applications AN440

```

AscHex:    CJNE    A,#'0',AH1      ; Test for ASCII numbers.
AH1:       JC      AHBAd           ; Is character less than a '0'?
          CJNE    A,#'9'+1,AH2     ; Test value range.
AH2:       JC      AHVal09        ; Is character is between '0' and '9'?

          CJNE    A,#'A',AH3      ; Test for upper case hex letters.
AH3:       JC      AHBAd           ; Is character is less than an 'A'?
          CJNE    A,#'F'+1,AH4     ; Test value range.
AH4:       JC      AHValAF        ; Is character is between 'A' and 'F'?

          CJNE    A,#'a',AH5      ; Test for lower case hex letters.
AH5:       JC      AHBAd           ; Is character is less than an 'a'?
          CJNE    A,#'f'+1,AH6     ; Test value range.
AH6:       JNC     AHBAd           ; Is character is between 'a' and 'f'?
          CLR     C
          SUBB    A,#27h           ; Pre-adjust character to get a value.
          SJMP    AHVal09         ; Now treat as a number.

AHBAd:     CLR     HexFlag         ; Flag char as non-hex, don't alter.
          SJMP    AHEx            ; Exit
AHValAF:   CLR     C
          SUBB    A,#7             ; Pre-adjust character to get a value.
AHVal09:   CLR     C
          SUBB    A,#'0'           ; Adjust character to get a value.
          SETB    HexFlag         ; Flag character as 'good' hex.
AHEx:      RET

; HexAsc - Convert a hexadecimal nibble to its ASCII character equivalent.

HexAsc:    ANL     A,#0Fh          ; Make sure we're working with only
          ; one nibble.
          CJNE    A,#0Ah,HA1       ; Test value range.
HA1:       JC      HAVal09         ; Value is 0 to 9.
          ADD     A,#7             ; Value is A to F, extra adjustment.
HAVal09:   ADD     A,#'0'          ; Adjust value to ASCII hex.
          RET

; ErrPrt - Return an error code to our host.

ErrPrt:    MOV     A,#':'          ; First, send a prompt that we are
          CALL    PutChar         ; still here.
          MOV     A,EFlags         ; Next, print the error flag value if
          JZ      ErrPrtEx        ; it is not 0.
          CALL    PrByte
ErrPrtEx:  RET

; CRLF - output a carriage return / line feed pair to the serial port.

CRLF:     MOV     A,#CR
          CALL    PutChar
          MOV     A,#LF
          CALL    PutChar
          RET

; PrByte - Send a byte out the serial port in ASCII hexadecimal format.

PrByte:    PUSH    ACC             ; Print ACC contents as ASCII hex.
          SWAP    A
          CALL    HexAsc           ; Print upper nibble.
          CALL    PutChar

```

RAM loader program for 80C51 family applications AN440

IEEE Micro Mouse using the 87C751 microcontroller AN443

Author: Tracy Ching

DESCRIPTION

Micro Mouse is an IEEE contest first proposed by the author of IEEE Spectrum in 1977. It consists of an autonomous robot known as a "mouse" which navigates through a maze of 256 two-inch-high, seven-inch squares in a 16×16 arrangement. The robot is self powered and has no knowledge of the maze configuration prior to releasing it in the maze. The first time it is released into the maze, its prime objective is to find a path from the starting square which is located in a corner of the maze to the destination square which can be located in the center or a different corner depending on competition level. The destination square for the advanced level can be found only by using a smart algorithm which will make the mouse gravitate towards the center without becoming lost. The destination square for a novice contest can be found by using a wall hugging algorithm. The analogy for this algorithm is to imagine a blind person holding their right hand out against a wall and following the walls until they reach the destination. A maximum of ten runs or 15 minutes is given to each mouse. Leaving the starting square constitutes one run. The mouse with the fastest run time from the starting square to the destination square wins.

The contest is held with different competition levels for novices and advanced. The novice level is typically held for college students trying to exercise newly acquired hardware and software skills, whereas the advanced level encompasses international talent and may require the implementation of a proportional integral derivative (PID) controller in software to utilize commutated or brushless DC motors and complex navigation algorithms.

DESIGN OBJECTIVES

Several design objectives were as follows: minimize weight, minimize part size and count, minimize power consumption which allows the use of smaller and lighter batteries, minimize cost and maximize speed.

The main objective was to keep the total weight at a minimum to obtain the fastest acceleration from the stepper motors and to reduce wheel slippage during deceleration and turning. The objectives are inter-related such that changing one will affect the other. Hence, having a low part count means lighter weight, faster acceleration and lower cost.

A typical mouse may consist of a microcontroller with supporting memory and GLU logic to interface the motor controllers

and sensors. The 87C751 is suitable for this application with its small size. The need for external RAM is eliminated by using the internal RAM to store minimum maze information suitable for the novice level. Nickel cadmium batteries are used because of the cost, size and power density obtainable versus other battery types. Nickel metal hydride was not available during development but would be a good choice over nickel cadmium batteries.

The 87C751 Microcontroller

The 87C751 is an 8-bit microcontroller based on the 8051 microcontroller family. It is code compatible with the exception of the MOVX, LJMP, and LCALL instructions. The MOVX instruction and external memory accesses are not supported. LJMP and LCALL instructions are not needed since AJMP and ACALL can reach the entire program memory range (2k bytes) of the 751. The 87C751 contains a $2k \times 8$ EPROM, a 64×8 RAM, 19 I/O lines, a 16 bit auto-reload counter/timer, a fixed rate timer, a five source fixed priority interrupt structure, a bidirectional Inter-Integrated circuit (I²C) bus interface, and an internal oscillator. The 87C751 comes in an erasable quartz package (87C751), one time programmable (87C751), and mask ROM (83C751).

HARDWARE DESCRIPTION

Figure 1 is a schematic diagram of the mouse. The stepper motors are 4 volts 0.95 amperes per coil giving about 14 oz-inches of torque. Each motor is driven by an Allegro UCN-5804B unipolar stepper motor translator/driver which contains the sequencing logic and high current darlington outputs. The sequencing logic only requires clock, direction of rotation, and output enable signals from the 87C751 which relieves the chore of having to cycle through a sequencing table for both motors therefore reducing code size. Fast recovery diodes are used to protect the darlington outputs from negative voltage due to motor winding flyback.

The infrared (IR) emitters are Optek OP-240A. The sensors are Optek OPL-560-OC which have a built-in light amplifier and TTL open collector output. Each sensor bank is enabled via a high side P-channel MOSFET. A 74LS04 (U4) is used to drive the P-channel MOSFET. Data from each bank of eight IR sensors is fed into port 3. By using open collector output sensors, the need for latching the data using a 3-state buffer is eliminated. Port pins P0.0 and P0.1 are used for enabling either the left or right

sensor bank via the 74LS04 (U4) and the MOSFETs. Each sensor bank hangs over the two inch high walls whereby the light emitted from the OP-240 is reflected into the OPL-560-OC if a wall is present. Two sensor pairs in the middle of the array are typically sensing the presence of a wall and the remaining sensors are used for guidance to keep the mouse running parallel to the walls.

A 74LS573 connected to port 3 which latched data into eight LEDs was used in the debugging process. P1.7 was used on the latch signal for the 74LS573. Since the eight LEDs and latch were not needed for the final product, they were removed thus reducing weight and battery drain. LEDs D1 and D2 were also used in the debugging process and are currently used for visual feedback when selecting options during operation.

Power for the digital circuitry is fed by an 8.4 volt NiCAD battery pack, packaged in a 9 volt battery case, via a 7805 regulator. The circuit can run constantly with the sensors taking "snapshots" of the walls intermittently for at least 30 minutes satisfying the 15 minute maximum requirement. Power for the motors is fed by four "A" size NiCAD cells which have enough energy to run the motors for 15 minutes before becoming useless at about 1 volt per cell.

TASK PRIORITIES

Several tasks take place during program execution which are as follows in order of importance: pulsing the stepper motors according to a velocity profile table, gathering sensor data, deciding whether to accelerate, decelerate, turn left or right.

In-line coding is used to avoid calling subroutines that cause the program counter to be pushed onto the stack and use valuable RAM for the turn decisions. Interrupt subroutines are an exception.

Interrupt timer 0 (T0) vectors to the routine which supplies a pulse to the stepper motor drivers. This is the most important task because the stepper motors require a smooth train of pulses in order to prevent jerky or sporadic motions. Thus, T0 must have the highest priority and must not be interrupted. Although timer 0 is a 16 bit timer, only eight bits are used with the higher byte set to FFH. T0 takes care of pulsing the left and right motor drivers at different times by using two external registers which are used as prescalers. Each motor can be assigned a different prescale value via the assigned register in order to step each motor at a different step rate.

IEEE Micro Mouse using the 87C751 microcontroller

AN443

The velocity profile table for T0 was designed using a spreadsheet program with visual graphing. This aided the ability to derive an exponential table for the fastest stepper motor acceleration using qualitative observations.

The first part of the in-line code contains the routine to accelerate the mouse from a stopped position. A pointer for each motor is incremented until the end of the velocity profile table is reached. During acceleration, the left and right sensors are strobed and stored after each step caused by T0.

The routine that strobes the sensors for wall data returns several results through the use of flags. The routine first stores the sensor data in registers. This information is used to determine if the sensor array or mouse is too far right, left, or aligned in a square and to set the corresponding flags. It also returns the presence of a front wall using the innermost sensor pairs. The deceleration routine is similar to the acceleration routine except the pointers decrement through the velocity profile table skipping a few values each time. Deceleration uses less steps than acceleration.

The routine which decides whether to continue acceleration, deceleration, or turn left or right keeps track of step count or position of the mouse within a square in order to store wall information at the proper time. After the previous routines have stored the data for the front, left, and right walls, a decision is obtained. If the mouse is not in a back-up mode, the wall information, status of the back-up flag and left/right algorithm flag forms an offset byte. If the mouse is in a back-up mode, the decision from above plus the previous decision on the stack is used to form the offset byte. This offset is stored in the accumulator. The decision which contains the direction to turn is obtained using the MOVC instruction which points to the table using the data pointer. The logic table is shown in Tables 1 and 2 below.

External interrupt 0 (INT0) vectors to the routine which brings the mouse to a halt. INT0 subroutine disables T0, sets output enable high on the stepper motor drivers, and sets the return address for the program counter to 0000H. This causes the mouse to restart program execution without losing the internal RAM.

**Table 1. Logic Table:
Non-Back-Up Mode**

(1)	(2)	(3)	(4)
R	NONE	R	push R onto stack
R	F	R	push R onto stack
R	R	S	push S onto stack
R	R, F	L	push L onto stack
R	L	R	push R onto stack
R	L, F	R	push R onto stack
R	L, R	S	ignore
R	L, R, F	180	turn on B-U flag
L	NONE	L	push L onto stack
L	F	L	push L onto stack
L	R	L	push L onto stack
L	R, F	L	push L onto stack
L	L	S	push S onto stack
L	L, F	R	push R onto stack
L	L, R	S	ignore
L	L, R, F	180	turn on B-U flag

NOTES:

Abbreviations used:

R = right

L = left

F = front

S = straight

B-U = back-up flag

stack = RAM used for storing decisions (not for the program counter)

Non back-up mode:

(1) wall hugging algorithm (user selected)

(2) walls surrounding present square

(3) direction to turn

(4) operations to perform

**Table 2. Logic Table:
Back-Up Mode**

(1)	(2)	(3)	(4)
R	R	R	push S onto stack, clear B-U
R	L	L	pop stack only
R	S	S	push L onto stack, clear B-U
L	R	R	pop stack only
L	L	L	push S onto stack, clear B-U
L	S	S	push R onto stack, clear B-U
S	R	R	push L onto stack, clear B-U
S	L	L	push R onto stack, clear B-U
S	S	S	pop stack only

NOTES:

Abbreviations used:

R = right

L = left

F = front

S = straight

B-U = back-up flag

stack = RAM used for storing decisions (not for the program counter)

Back-up mode:

(1) previous decision from top of stack

(2) decision from the above table for this current square

(3) direction to turn (should be equal to 2)

(4) operations to perform (all operations require lowering the stack by one before pushing data)

OPERATION

Reset

True reset is accomplished only during power on. After completing the maze, the operator brings the mouse to a stop using the start/stop switch which effectively disables the motors and resets the program counter to 0000H, restarting the mouse's program with the exception that the RAM contains vital maze information.

Starting/Stopping

After power-up, the mouse remains idle with the motors and sensors disabled, awaiting an interrupt from switch SW1. The first time the switch is pressed, the mouse will start. The second time the switch is pressed, the mouse will stop and the cycle repeats.

IEEE Micro Mouse using the 87C751 microcontroller AN443

Selecting right or left wall hugging

After power on reset, the program defaults to right wall hugging. Pressing switch SW2 will cause the mouse to follow the left walls. If it is pressed again, it will follow the right walls—toggling between the left and right wall hugging algorithm each time it is pressed.

Increasing motor speed

After completing the first run, pressing switch SW2 causes the timer value to increase by an index of one, increasing the speed of the motors. This allows the mouse to be run at increasing speeds each run in order to attain the fastest possible time before crashing into a wall or incurring a condition in the stepper motors called "pull-out". Pull-out is a condition wherein the fields are changing in the coils faster than the rotor can maintain synchronism with the coils, causing the rotor to stall.

SOFTWARE LIMITATIONS

The 87C751 has 64 bytes of RAM. The program requires 31 bytes of RAM leaving 33 bytes for storing decisions. One hundred thirty two decisions can be stored in 33 bytes

of RAM since four decisions are stored per byte. Although there are 16 times 16 squares with possibly 256 decisions to be made, this condition is not very probable since every square typically is not made into a turning point. Long straight ways are used as well as turning points. The probability that the RAM will fill to the maximum capacity is very slim in the novice level where there are typically 50 decisions to be made. Hence, RAM which can hold 132 decisions is adequate but not infallible.

FUTURE ENHANCEMENTS

Although the 87C751 has minimal RAM, I²C RAM could be easily added by switching the MOSFET drivers using the LED port pins and placing the I²C RAM on port pins P0.0 and P0.1. This would allow the implementation of a full mapping algorithm thus allowing entry to the advanced class. The code size for the I²C routines and recursive algorithm to find the center of the maze would add about 800 bytes more of code. Since some of the code for the novice class would be eliminated, it would bring the total code size to less than 2k bytes. A few maze solving algorithms have been implemented successfully on other

mice. These are the flooding or Bellman's algorithm, backtracking algorithm and others. The first two algorithms are recursive, thus the code sizes are quite small.

Another added feature would be the ability to execute a rounded turn rather than pivot. This requires a more complex navigation scheme and velocity profiler to make the motors turn at differing speeds in order to make the rounded turn.

In addition to the enhancements aforementioned, methods to track a wall through the use of an A/D converter which measures the reflectivity strength from infrared sensors pointing at the sides of the walls can also be implemented on the I²C bus. This would eliminate the array of sensors hanging in front of the mouse on top of the walls thus decreasing the weight.

CONCLUSION

The 87C751 microcontroller provides the required computing resources and I/O ports necessary to control a robotics device known as a "Micro Mouse". The I²C interface allows for an abundance of variations on this robotics device.

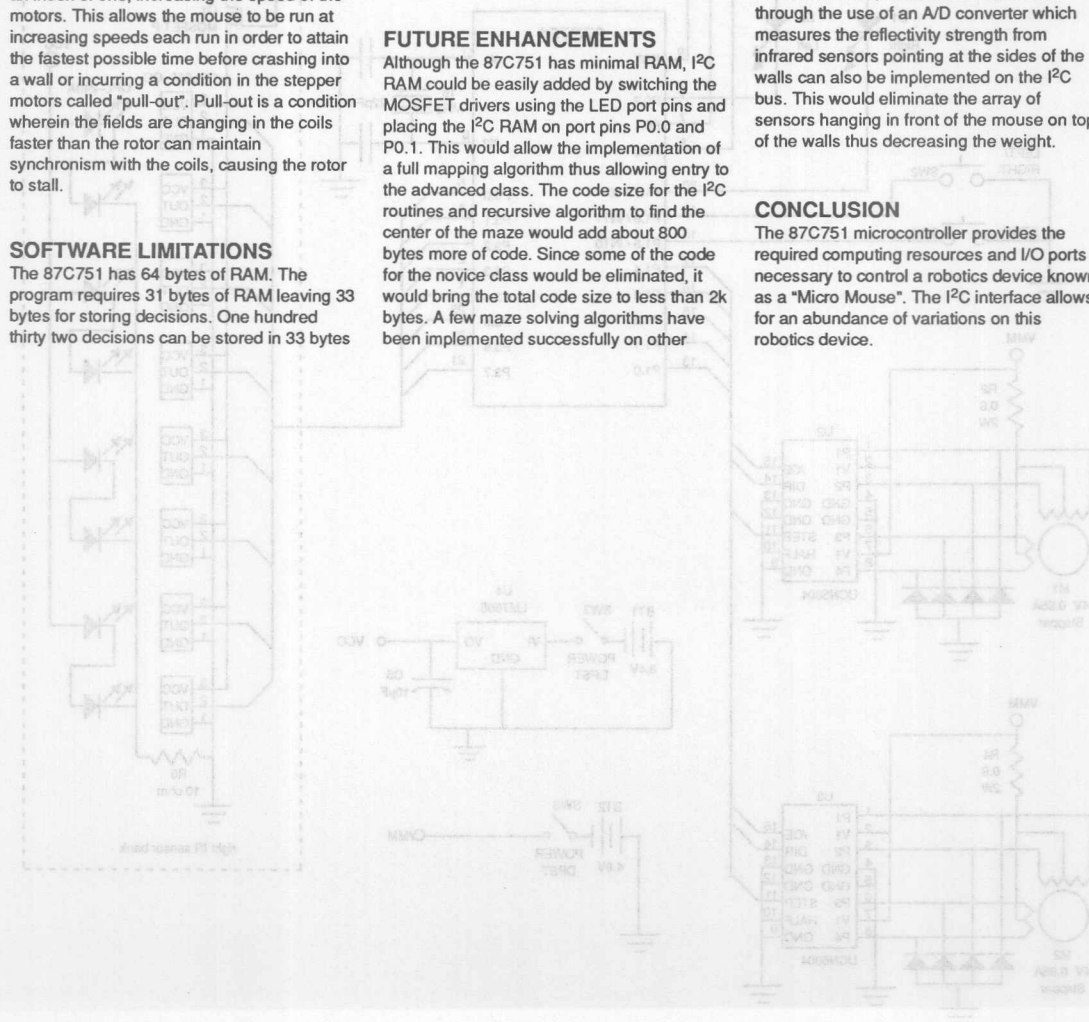


Figure 1. Schematic Diagram of the Micro Mouse

IEEE Micro Mouse using the 87C751 microcontroller AN443

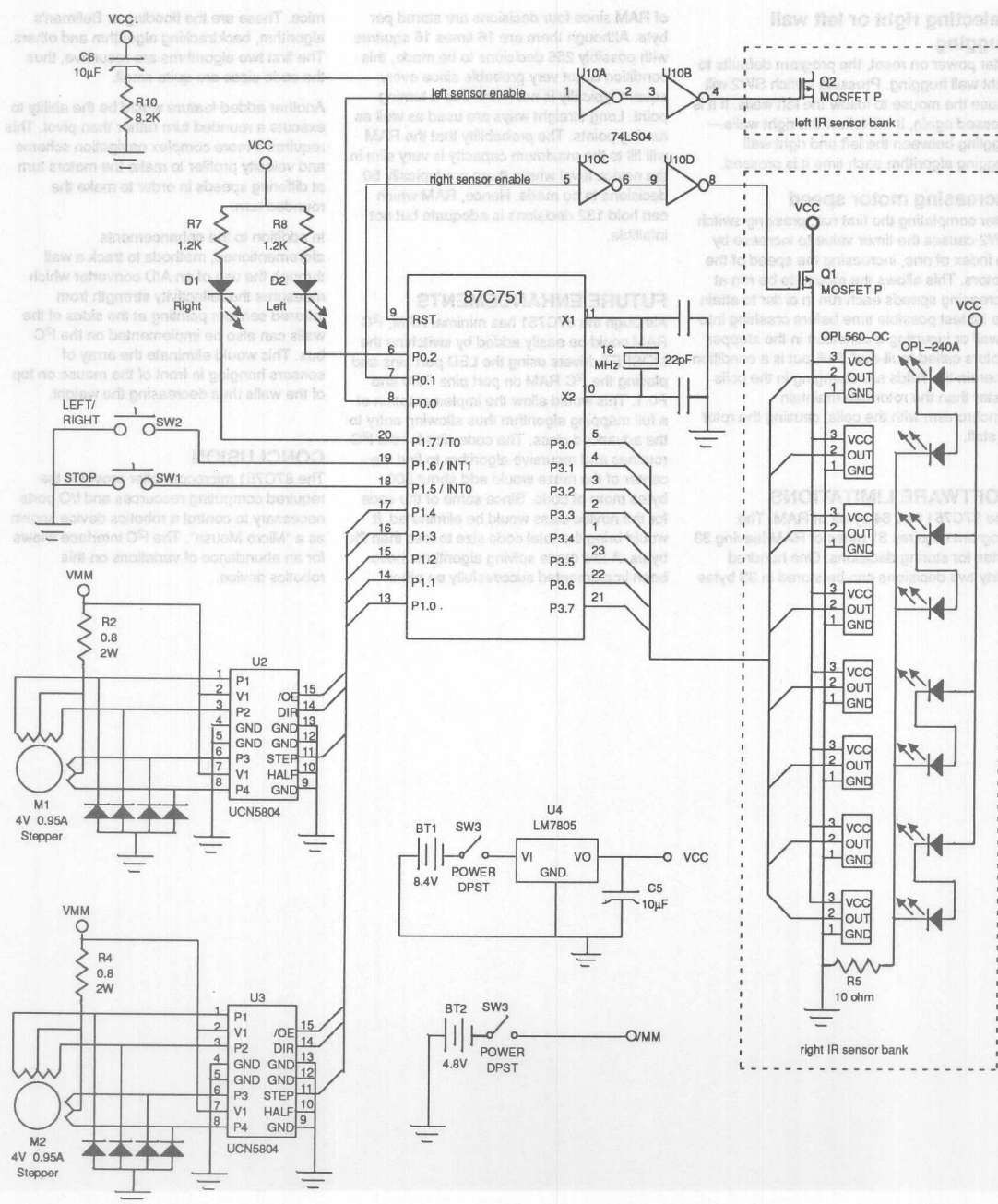


Figure 1. Schematic Diagram of the Maze Mouse

IEEE Micro Mouse using the 87C751 microcontroller AN443

MCS-51 MACRO ASSEMBLER 80 751MAIN

04/16/92 PAGE 12-224

```

LOC OBJ      LINE      SOURCE
1 1 ;*****
2 2 ; 87C751 Micro Mouse version 4.0 started 9-27-91 *
3 3 ; version 4.3 finished 4-16-92 *
4 4 ; Algorithms and programming by Tracy Ching *
5 5 ; Some symbols commented out for use as in-line coding *
6 6 ; rather than being used as subroutines (S.R.) *
7 7 ;*****
8 8
9 +1 $include (751.equ)
=1 10 ;These are all RAM addresses
=1 11 ;equate table of constants
=1 12 ;ind_reg equ 00 ;used for ind addr of regs
=1 13 ;map_org equ 01 ;pointer for storing map of maze
=1 14 ; equ 02 ;decide storage-local,
=1 15 ; equ 03 ;do180, turn90 use it
=1 16 ; equ 04 ;snapshot R wall storage
=1 17 ; equ 05 ;snapshot L wall storage
=1 18 ; equ 06 ;store_val, decel, int 1, gen purp
=1 19 ; equ 07 ;pause setting, gen purp
=1 20
0008 =1 21 step_count equ 08h ;step count used by decide
0009 =1 22 temp_reg1 equ 09h ;
000A =1 23 ls373 equ 0ah ;storage for 74LS373 debug data leds
000B =1 24 count_fw equ 0bh ;count for wall, step count
000C =1 25 l_timr equ 0ch ;value to be used in the isr
000D =1 26 r_timr equ 0dh ;
000E =1 27 l_ptr equ 0eh ;points at accel table
000F =1 28 r_ptr equ 0fh ;
0010 =1 29 map_offset equ 10h ;points to the specific two bits in map ptr
=1 30 ; equ 11h
=1 31 ; equ 12h
=1 32 ; equ 13h
=1 33 ; equ 14h
=1 34 ; equ 15h
=1 35 ; equ 16h
=1 36 ; equ 17h
=1 37
=1 38 ;PCON EQU 87H
=1 39 ;TA EQU 0C7H
008B =1 40 rtl equ 8bh
008D =1 41 rth equ 8dh
=1 42
=1 43
=1 44 ;flag declarations, they are erased after every restart
0020 =1 45 ss_bits equ 20h ;sens flag reg
0000 =1 46 too_r bit 20h.0 ;too close to right wall
0001 =1 47 too_l bit 20h.1 ; left wall
0002 =1 48 r_wall bit 20h.2 ;right wall present, snapshot storage
0003 =1 49 l_wall bit 20h.3 ;left wall present, snapshot storage
0004 =1 50 f_wall bit 20h.4 ;front wall present, snapshot storage

```

IEEE Micro Mouse using the 87C751 microcontroller AN443

MCS-51 MACRO ASSEMBLER 751MAIN

04/16/92 PAGE 24

```

LOC OBJ      LINE      SOURCE
0005          =1 51 aligned bit 20h.5 ;sensors detect OK straightness
0006          =1 52 far_r bit 20h.6 ;used to sync step count
0007          =1 53 far_l bit 20h.7 ;ditto - left
0008          =1 54
0008          =1 55 r_turn bit 21h.0 ;decide sets this for turn S.R.
0009          =1 56 l_turn bit 21h.1 ;ditto
000A          =1 57 s_s_int bit 21h.2 ;start stop bit
000B          =1 58 time_int bit 21h.3 ;timer 0 int has occurred
000C          =1 59 r_int bit 21h.4 ;r motor was stepped
000D          =1 60 l_int bit 21h.5 ;l mot
000E          =1 61 slow_r bit 21h.6 ;make int_0 incr timer flag
000F          =1 62 slow_l bit 21h.7 ;ditto left
0010          =1 63
0010          =1 64 get_out bit 22h.0 ;used to flag accel to end so decel can do
0011          =1 65 r_decide bit 22h.1 ;r wall present, decide storage
0012          =1 66 l_decide bit 22h.2 ;l wall present, decide storage
0013          =1 67 f_decide bit 22h.3 ;f wall present, decide storage
0014          =1 68 make_180 bit 22h.4 ;decide says do 180
0015          =1 69 prev_r bit 22h.5 ;decide uses for detect wall transition
0016          =1 70 prev_l bit 22h.6 ;ditto left
0017          =1 71 cw180 bit 22h.7 ;toggle dir of 180
0018          =1 72
0018          =1 73 det_L2H bit 23h.0 ;used to detect the low to high wall
0019          =1 74 look4f bit 23h.1 ;skips to look for front wall
001A          =1 75 back_up bit 23h.2 ;tells decide that it is backing up
001B          =1 76 look bit 23h.3 ;start looking at wall
001C          =1 77 count_en bit 23h.4 ;enables step counting
001D          =1 78 skip_decide bit 23h.5 ;TEMPORARY (testing only)
001E          =1 79 genp2 bit 23h.6 ;chk R or L sens flag
001F          =1 80 genp1 bit 23h.7 ;gen purpose, watch for conflicts betwx S.R.s
001F          =1 81
001F          =1 82
001F          =1 83
001F          =1 84 ;these are the flags that don't get erased after restarting
0020          =1 85 done bit 24h.0 ;tells the prog that it has gone thru
0021          =1 86 l_r_bit bit 24h.1 ;hug left or right bit, set=left, clr=right
0022          =1 87 temp_bit1 bit 24h.2 ;local bit var
0022          =1 88
0022          =1 89
0022          =1 90 ;hardware definitions
0082          =1 91 l_led bit p0.2
0097          =1 92 r_led bit p1.7
0090          =1 93 mot_en bit p1.0
0081          =1 94 r_sens bit p0.1
0080          =1 95 l_sens bit p0.0
0092          =1 96 r_step bit p1.2
0091          =1 97 r_dir bit p1.1
0094          =1 98 l_step bit p1.4
0093          =1 99 l_dir bit p1.3
0095          =1 100 s_s_sw bit p1.5
0096          =1 101 l_r_sw bit p1.6
00B0          =1 102 sensors equ p3
00B0          =1 103 +1 $include (extrn751.tab)
00B0          =1 104 ;These are all of the mouse constants.
00B0          =1 105

```

MCS-51 MACRO ASSEMBLER 80 751MAIN

DATE 04/16/92 PAGE 32-230

LOC	OBJ	LINE	SOURCE	LOC	OBJ	LINE	SOURCE
		=1 106	extrn number (timer_reload)				
		=1 107	extrn number (pivot_speed)				
		=1 108	extrn number (acc_steps)				
		=1 109	extrn number (half_way)				
		=1 110	extrn number (hold_val)				
		=1 111	extrn number (dec_acc)				
		=1 112	extrn number (decel_steps)				
		=1 113	extrn number (decel_decr)				
		=1 114	extrn number (steps90)				
		=1 115	extrn number (half_90)				
		=1 116	extrn number (look90)				
		=1 117	extrn number (steps180)				
		=1 118	extrn number (half_180)				
		=1 119	extrn number (look180)				
		=1 120	extrn number (map_org_addr)				
		=1 121	extrn number (sens_pat)				
		=1 122	extrn number (pause_val)				
		=1 123	extrn number (w2nw_cnt)				
		=1 124	extrn number (chk4fr)				
		125					
0000		126	org 00				
0000 0115		127	ajmp main				
0003		128	org 03				
0003 618D		129	ajmp int_0				
000B		130	org 0bh				
000B 6123		131	ajmp tim_0				
0013		132	org 13h				
0013 6164		133	ajmp int_1				
		134					
		135					
		136	;				
		137	; This section initializes the registers. Note that the maze info				
		138	; is not cleared.				
		139	;				
		140					
0015 C220		141	main: clr done				
0017 758B00	F	142	mov r1, #timer_reload				
001A 758DFF		143	mov r1, #0ffh				
		144					
001D D281		145	main_1: setb r_sens				
001F D280		146	setb l_sens				
0021 43907F		147	orl p1, #7fh				
0024 752000		148	mov r0, #0				
0027 752100		149	mov r1, #0				
002A 752200		150	mov r1, #22h, #0				
002D 752300		151	mov r1, #23h, #0				
0030 7810		152	mov r0, #10h				
0032 7600		153	main_0: mov @r0, #0				
0034 D8FC		154	djnz r0, main_0				
		155	; initialize values				
0036 758125		156	mov sp, #25h				
0039 7900	F	157	mov r1, #map_org_addr				
003B 75A887		158	mov ie, #1000011b				
003E 300AFD		159	jnb s_s_int, \$				
		160					

IEEE Micro Mouse using the 87C751 microcontroller AN443

MCS-51 MACRO ASSEMBLER \80 751MAIN

04/16/92 PAGE 44

```

LOC  OBJ          LINE    SOURCE
161
162 ;*****
163 ; This section accelerates the mouse thru the velocity table.      *
164 ; The table value is lowered if the mouse is too close to a wall.  *
165 ;*****
166
167 accel:                ;loads acc_table and goes from there
0041 71B7             168      acall snapshot
0043 A202             169      mov  (0c,r_wall      ;check the walls
0045 9215             170      mov  (0prev_r,c      ;
0047 A203             171      mov  (c,l_wall      ;for both sides
0049 9216             172      mov  (00prev_l,c    ;
004B 750841          173      mov  (0step_count,#65 ; start at 75steps, sens mid way
004E 750E00          174      acc_aa:  mov  (0l_ptr,#0
0051 750F00          175      mov  (0r_ptr,#0      ;clear offsets
0054 750DFE          176      mov  (0ar_tmr,#0feh ;even up the timers to step
0057 750CFE          177      mov  (00l_tmr,#0feh ;evenly
005A D28C             178      setb (0tr0 ;enable timer int
179
005C 300BFD          180      acc_0:  jnb   time_int,$      ;stay here until int happens
005F C20B             181      clr   00 time_int
0061 71B7             182      acall snapshot      ;get status of too_l too_r
183
184      ;right routine
0063 300C21          185      jnb  #00 r_int,acc_1      ;skip if right hasn't been stepped yet
0066 C20C             186      clr  #0 r_int           ;ackn r int
0068 E50F             187      mov  #0 a,r_ptr         ;check to see if at end of acc
006A B40002 F        188      cjne a,#acc_steps,acc_1c
006D 0171             189      ajmp  acc_1b
006F 050F             190      acc_1c:  inc   r_ptr           ; point to next accel value
0071 300113          191      acc_1b:  jnb  too_l,acc_1      ;if not too left then fall thru
0074 C3              192      clr   c              ;subtr halfway from ptr
0075 E50F             193      mov  a,r_ptr
0077 9400 F          194      subb  a,#half_way      ;C set if ptr < half_way
0079 E50D             195      mov  a,r_ptr
007B 4006             196      jc   acc_1a
007D 2400 F          197      add  #1 a,#hold_val      ; slow down even more
007F F50D             198      mov  #0 r_tmr,a          ;load it back into timer decel value
0081 0187             199      ajmp  acc_1
0083 2400 F          200      acc_1a:  add  #1 a,#dec_acc      ;dec_acc used when going fast
0085 F50D             201      mov  #0 r_tmr,a          ;load it back into timer decel value
202
203      ;left routine
0087 300D21          204      acc_1:  jnb  #00 l_int,acc_2      ;if L hasn't been stepped, skip
008A C20D             205      clr  #00 l_int           ;ackn l int
008C E50E             206      mov  #0 a,l_ptr         ;check to see if at end of acc
008E B40002 F        207      cjne a,#acc_steps,acc_2c
0091 0195             208      ajmp  acc_2b
0093 050E             209      acc_2c:  inc  #0 l_ptr          ; point to next accel value
0095 300013          210      acc_2b:  jnb  too_r,acc_2      ;if not too right then fall thru
0098 C3              211      clr   c              ;subtr halfway from ptr
0099 E50E             212      mov  #0 a,l_ptr
009B 9400 F          213      subb  a,#half_way      ;C set if ptr < half_way
009D E50C             214      mov  #0 a,l_ptr
009F 4006             215      jc   acc_2a

```

IEEE Micro Mouse using the 87C751 microcontroller AN443

MCS-51 MACRO ASSEMBLER 751MAIN

04/16/92 PAGE 52

```

LOC OBJ      LINE      SOURCE
00A1 2400     216      addl a,#hold_val      ; slow down even more
00A3 F50C     217      movl _l_timr,a      ;load it back into timer decel value
00A5 01AB     218      ajmp acc_2      ;get out
00A7 2400     219      acc_2a: addl a,#dec_acc      ;dec_acc used when going fast
00A9 F50C     220      movl _l_timr,a      ;load it back into timer decel value
00AB 201853   221      movl _l_timr,a      ;load it back into timer decel value
00AC 201853   222      acc_2:      ;
00AD 201853   223      ;
00AE 201853   224      ;
00AF 201853   225      ;*****
00B0 201853   226      ;This section is the "decision-maker". It makes the decision to *
00B1 201853   227      ; quit acceleration, turn left or right, make a 180 pivot, store
00B2 201853   228      ; a decision, or get a decision from the stack.
00B3 201853   229      ;*****
00B4 201853   230      ;
00B5 201853   231      ;decide:
00B6 201853   232      ;r_turn, l_turn, get_out, make_180 as public bits
00B7 201853   233      ;r_decide, l_decide, f_decide as local bits
00B8 201853   234      ;This S.R. decides when to start the decel using a step count
00B9 201853   235      ;when it sees a front wall or when it is time to turn. It also
00BA 201853   236      ;decides which way to turn on a mapping run or a final run.
00BB 201853   237      ;1) look for a wall to no wall transition
00BC 201853   238      jnb _l2H,di_3_1 ;skip the stuff below if set
00BD 201853   239      movl c,prev_r
00BE 201853   240      anl _l2H,c,r_wall ;check for r wall transition
00BF 201853   241      jnc di_1 ;if no transition goto di_1
00C0 201853   242      movl _l2H,step_count,#0 ;start counting
00C1 201853   243      di_1:      movl c,prev_l
00C2 201853   244      anl _l2H,c,l_wall ;check for l wall transition
00C3 201853   245      jnc di_2 ;if no transition goto di_2
00C4 201853   246      movl _l2H,step_count,#0 ;start counting
00C5 201853   247      di_2:      ;
00C6 201853   248      ;2) now detect step count
00C7 201853   249      movl a,step_count ;check to see if
00C8 201853   250      clrl _l2H,c ;
00C9 201853   251      subb _l2H,a,#84 ; 80 <= steps <= 83
00CA 201853   252      jnc di_3
00CB 201853   253      movl a,step_count
00CC 201853   254      clrl _l2H,c ;
00CD 201853   255      subb _l2H,a,#80
00CE 201853   256      jnc di_3
00CF 201853   257      movl a,c,r_wall ;store R wall into decide storage
00D0 201853   258      movl _l2H,r_decide,c ; for use later in CASE
00D1 201853   259      movl a,c,l_wall ;
00D2 201853   260      movl _l2H,l_decide,c ; when CASE statment is built below
00D3 201853   261      clrl _l2H,skip_decide
00D4 201853   262      di_2_0:      ajmp _l2H,di_9 ;get out since did above stuff
00D5 201853   263      ;
00D6 201853   264      di_3:      ;3) detect no wall to wall (lo to hi) and check for front wall
00D7 201853   265      ;count must be greater than 120 to allow for spaces in posts
00D8 201853   266      movl a,step_count
00D9 201853   267      clrl _l2H,c
00DA 201853   268      subb _l2H,a,#120
00DB 201853   269      jnc di_3_1
00DC 201853   270      movl c,prev_r ;the prev should be low

```


IEEE Micro Mouse using the 87C751 microcontroller AN443

MCS-51 MACRO ASSEMBLER 751MAIN

04/16/92 PAGE 6

```

LOC  OBJ          LINE    SOURCE
00E3  A002          271      ;check for r wall transition
00E5  400A          272      ;IF no lo to hi then di_3_0
                                273      ;found lo to hi, set step counter
00E7  750800        274      mov step_count,#w2nw_cnt
00EA  C213          275      clr f_decide
00EC  750B00        276      mov count_fw,#0 ;clr count for use in decel S.R.
00EF  D218          277      setb det_L2H
00F1  A216          278      di_3_0: mov c,prev_1 ;the prev should be low
00F3  A003          279      orl c,/l_wall ;check for l wall transition
00F5  400A          280      jc di_3_1 ;IF no lo to hi then di_3_1
                                281      ;found lo to hi, set step counter
00F7  750800        282      mov step_count,#w2nw_cnt
00FA  C213          283      clr f_decide
00FC  750B00        284      mov count_fw,#0
00FF  D218          285      setb det_L2H
                                286      di_3_1: ;condition that will get you to di_3_9
                                287      ;det_L2H & count > 162
0101  A204          288      movl acc,f_wall ; trigger from extrn lite during
0103  9213          289      mov acc,f_decide,c ; other points in square
0105  400E          290      jc di_3_9 ;get out if meet condition ELSE cntnu
0107  3018CE        291      jnb det_L2H,di_2_0 ;this section checks second cond
                                292
010A  E50B          293      mov a,count_fw
010C  B40002        294      cjne a,#chk4fr,di_3_2
010F  2115          295      ajmp di_3_9
0111  050B          296      DI_3_2: inc count_fw ;
0113  2197          297      ajmp di_9 ;get out
                                298
                                299      di_3_9: ;at this point we have detected it's time to stop, store in RAM, and
                                300      ;reexecute plan... Therefore set bit to skip all of this
0115  201DC0        301      jb skip_decide,di_2_0
0118  D21D          302      setb skip_decide
                                303
011A  C218          304      clr det_L2H
011C  750800        305      mov step_count,#0
011F  202054        306      jb done,dif_0 ;IF done THEN dif_0 ELSE fall thru
                                307
                                308      di_4: ;4) at this point F L R are loaded, set up CASE table
                                309      ; L=01 R=10 S=11 stop-mouse=00 all represented as two bits.
0122  7400          310      mov a,#0
0124  A221          311      movl a,c,l_r_bit ;if no back up then we have this...
0126  33          312      rlc a ; |
0127  A212          313      movl a,c,l_decide ; |
0129  33          314      rlc a ; | L/R_bit 1=L
012A  A211          315      movl a,c,r_decide ; | 0=R
012C  33          316      rlc a ; |
012D  A213          317      movl a,c,f_decide ; |
012F  33          318      rlc a ;| 0 | 0 | 0 | 0 | L/R | L | R | F |
0130  9001A1        319      movl a,dptr,#d_table ;point to decision table
0133  93          320      movl a,@dptr ;get decision
0134  FA          321      movl a,r2,a ;save it for later
                                322
0135  201A14        323      jnb back_up,di_4_0 ;IF backing up GOTO di
                                324      ;5) execute the table from here. | A | part is taken care of here
0138  A2E7          325      movl a,c,acc.7 ;setting make_180 flag

```


IEEE Micro Mouse using the 87C751 microcontroller AN443

MCS-51 MACRO ASSEMBLER \40 751MAIN

04/16/92 PAGE 04 72-004

LOC	OBJ	LINE	SOURCE	SOURCE	LINE	LOC
013A	9214	326	mov dx,make_180,c		18E	013B 9215
013C	9210	327	mov dx,get_out,c		18E	013D 9216
013E	921A	328	mov dx,back_up,c		18E	013F 9217
0140	4055	329	jnc dx,di_9	;if make_180 then return		
		330		;this determines D direction to turn		
0142	5403	331	anl a,#00000011b	;mask out junk to get D		
0144	F5F0	332	mov b,a			
0146	31E3	333	acall store_val	;store the decision and then store		
0148	31C1	334	acall inc_map_ptr	; a halt afterwards and decre		
		335				
014A	802E	336	sjmp dx,58	;go execute it below		
		337				
014C	31D2	338	di_4_0:			
014E	5117	339	acall get_val	;get top of stack, put in B		
0150	EA	340	mov a,r2			
0151	5403	341	anl a,#00000011b	;keep D (dir to turn)		
0153	C5F0	342	xch a,b	;stack top in A, D in B		
0155	23	343	rl a	;stack top now looks like this		
0156	23	344	rl a	; 0000XX00		
0157	45F0	345	orl a,b	;looks like 0000 prev D		
0159	D2E4	346	setb acc.4	;looks like 000 B/U prev D		
		347				
015B	93	348	movc a,@a+dptr	;get decision		
015C	FA	349	mov r2,a	;save it just in case		
015D	C214	350	clr make_180			
015F	A2E6	351	mov c,acc.6	;setting or clearing back-up		
0161	921A	352	mov back_up,c			
0163	201A14	353	jb back_up,dx_58	;if still backing, dont store		
0166	C4	354	swap a			
0167	5403	355	anl a,#00000011b	; C now in B		
0169	F5F0	356	mov b,a			
016B	31E3	357	acall store_val	; and now stored in RAM		
016D	31C1	358	acall inc_map_ptr			
016F	EA	359	mov a,r2	;get decision and mask to get		
0170	5403	360	anl a,#00000011b	; D dir to turn		
0172	F5F0	361	mov b,a			
0174	8004	362	sjmp dx_58	;go execute it below		
		363				
		364		;6) skip all of the junk above and do this if running finals		
		365	dif_0:	;do this if running final solution		
0176	5117	366	acall get_val	;get the turn decision		
0178	31C1	367	acall inc_map_ptr	;incr map pointer		
		368				
		369				
		370		;7) interpret A to set the turn flags accordingly		
		371	dx_58:	;r_turn, l_turn flags		
017A	E5F0	372	mov a,b	;cjne's only work on acc not b		
017C	B40108	373	cjne a,#01,dx_52	;look for		
017F	D208	374	setb r_turn			
0181	C209	375	clr l_turn			
0183	D210	376	setb get_out			
0185	8010	377	sjmp dx_52			
0187	B40208	378	dx_52:	cjne a,#02,dx_53		
018A	C208	379	clr r_turn			
018C	D209	380	setb l_turn			

```

LOC OBJ          LINE    SOURCE
018E D210        381      c, setb r10, get_out
0190 8005        382      c, jmp stop di_9
                        383      dx_53: c, must be b=11 or b=00, straight or stop
                        384      ;if 00 then stop or HALT
0192 B40002      385      cjne r10, a, #0, di_9
0195 8153        386      ajmp halt
                        387
                        388      ;end of S.R. here at di-9
0197 A202        389      mov c, r_wall
0199 9215        390      mov prev_r, c      ;store the wall condition for next
019B A203        391      movb dx_53, c_l_wall ; toggle look
019D 9216        392      mov prev_l, c
                        393
019F 412C        394      ajmp decide_x
                        395
                        396      ;*****
                        397      d_table: ;this holds all of the answers for the decision table
                        398      ;byte form looks like this... | A | B | C | D | (two bits ea)
                        399      ; A - XY, X=do 180, Y=B-U flag status
                        400      ; B - add to stack if in B-U, L=10 R=01 S=11 stop-mouse=00
                        401      ; C - what to add to stack not in B-U (B-U means back-up mode)
                        402      ; D - direction to turn
01A1 05          403      db 00000101b, 00000101b, 00001111b, 00001010b
01A2 05          404
01A3 0F          405
01A4 0A          406
01A5 05          407      db 00000101b, 00000101b, 00001111b, 11001111b
01A6 05          408
01A7 0F          409
01A8 CF          410
01A9 0A          411      db 00001010b, 00001010b, 00001010b, 00001010b
01AA 0A          412
01AB 0A          413
01AC 0A          414
01AD 0F          415      db 00001111b, 00000101b, 00001111b, 11001111b
01AE 05          416
01AF 0F          417
01B0 CF          418
                        419      ;these lines are for the B-U mode decisions, some non-valid
01B1 00          420      db 00h, 00h, 00h, 00h
01B2 00          421
01B3 00          422
01B4 00          423
01B5 00          424
01B6 31          425      db 00h, 001100001b, 01000010b, 00100011b
01B7 42          426
01B8 23          427
01B9 00          428      db 00h, 01000001b, 00110010b, 00010011b
01BA 41          429
01BB 32          430
01BC 13          431
01BD 00          432      db 00h, 00100001b, 00010010b, 01000011b
01BE 21          433
01BF 12          434
01C0 43          435

```

IEEE Micro Mouse using the 87C751 microcontroller AN443

MCS-51 MACRO ASSEMBLER 80 751MAIN

04/16/92 PAGE 9 2-20N

```

LOC  OBJ          LINE    SOURCE
412
413      ; 0 0 0 0 | L/R | L | R | F |      for normal address
414      ; 0 0 0 0 | 0 | B/U | PREV |WAY_2_GO|      for back-up address
415      ; map representation in RAM will look like this...
416      ;*****
417      ;map representation in RAM will look like this...
418      ;most sig 2 bits| XX | XX |least sig 2 bits|
419      ;location = | D | C | B | A | where A is the first decision, B is second
420      ;etc. A, B, C, D are all two bits. The next byte of RAM is rep the same.
421      ; It requires R1 as pointer and MAP_OFFSET 10h
422      ; L=10 R=01 S=11 stop-mouse=00 all represented as two bits.
423      ; If pointer = 3Fh and map offset=4 then end of RAM
424      ;*****
425      inc_map_ptr:      ;this S.R. incs the map pointer from 0 to 3 and map_org
426      ;MAP uses mem-addr 11h-1fh and 2ch-3fh. R1 holds addr.
427
01C1 AF10      428      mov     r7,map_offset      ;cjne only works on local regs
01C3 BF0309      429      cjne    r7,#3,imp_0      ;IF 3 THEN do below ELSE imp_0
01C6 7510FF      430      mov     map_offset,#0ffh      ;start at MSB two bits (offset=0)
01C9 B91F02      431      cjne    r1,#1fh,imp_1      ;
01CC 792B      432      mov     r1,#2bh
01CE 09      433      inc     r1      ; and also move pointer high
01CF 0510      434      inc     map_offset      ;
01D1 22      435      ret
436
437      ;*****
438      dec_map_ptr:      ;this S.R. decr the map pointer and returns what's on the
439      ; top of the stack in B
440
01D2 AF10      440      mov     r7,map_offset      ;cjne only works on local regs
01D4 BF0009      441      cjne    r7,#0,dmp_0      ;IF 0 THEN do below ELSE dmp_0
01D7 751004      442      mov     map_offset,#4      ;point |XX|XX|XX|00| at 00
01DA B92C02      443      cjne    r1,#2ch,dmp_1
01DD A920      444      mov     r1,20h
01DF 19      445      dmp_1:      dec     r1      ;map pointer R1
01E0 1510      446      dmp_0:      dec     map_offset
01E2 22      447      ret
448
449      ;*****
450      store_val:      ;requires the decision to be in B as 000000XX and pointed to
01E3 C7      451      xch     a,@R1      ;get byte for below but save a
01E4 AE00      452      mov     r6,b      ;decision is in B
01E6 AF10      453      mov     r7,map_offset      ;get it
01E8 BF0002      454      cjne    r7,#0,sv_a      ;are we at 0 yet?
01EB 8004      455      sjmp     sv_b      ; I guess we are
01ED 03      456      sv_a:      rr     a      ;roll bits to shift |00|XX|00|00|
01EE 03      457      rr     a      ;to get |XX|00|00|00|
01EF DFFC      458      djnz    r7,sv_a      ;keep shifting til |00|00|00|XX|
01F1 C2E0      459      sv_b:      clr     acc.0      ;clear it out to load it below
01F3 C2E1      460      clr     acc.1
01F5 BE0104      461      cjne    r6,#01,sv_0      ;load R if 01
01F8 D2E0      462      setb    acc.0      ;looks like |XX|XX|XX|01|
01FA 800E      463      sjmp     sv_2
01FC BE0204      464      sv_0:      cjne    r6,#02,sv_1      ;load L if 02
01FF D2E1      465      setb    acc.1      ;looks like |XX|XX|XX|10|
0201 8007      466      sjmp     sv_2

```

IEEE Micro Mouse using the 87C751 microcontroller

MCS-51 MACRO ASSEMBLER 751MAIN

04/16/92 PAGE 10

```

LOC OBJ      LINE      SOURCE
0203 BE0304   467      sv_1:      cjne      r6,#03,sv_2
0206 D2E0     468      setb      acc.0      ;load Straight AKA S
0208 D2E1     469      setb      acc.1      ;looks like |XX|XX|XX|11|
020A AF10     470      sv_2:      mov      r7,map_offset ;roll until bits in original position
020C BF0002   471      cjne      r7,#0,sv_c      ;are we at 0 yet?
020F 8004     472      sjmp      sv_d      ; I guess we are
0211 23       473      sv_c:      rl      a      ;roll bits to shift |00|00|00|XX|
0212 23       474      rl      a      ;to get |XX|00|00|00|
0213 DFFC     475      djnz      r7,sv_c      ;keep shifting til |00|XX|00|00|
0215 C7       476      sv_d:      xch      a,@r1      ;put it all back
0216 22       477      ret
0217 F509     481      get_val:      ;this S.R. returns turn value in B
0219 E7       482      mov      temp_reg1,a      ;save acc same as push acc
021A AF10     483      mov      a,@r1
021C BF0002   484      cjne      r7,#0,gv_0      ;get it
021F 8004     485      sjmp      gv_1      ; I guess we are
0221 03       486      gv_0:      rr      a      ;roll bits to shift |00|00|00|XX|
0222 03       487      rr      a      ;to get |00|00|XX|00|
0223 DFFC     488      djnz      r7,gv_0      ;keep shifting til |XX|00|00|00|
0225 5403     489      gv_1:      anl      a,#00000011b ;now have |00|00|00|XX|
0227 F5F0     490      mov      b,a      ;stuff it back
0229 E509     491      mov      a,temp_reg1
022B 22       492      ret
022C 101002   497      decide_x:      jbc      get_out,decel ;get out, time to stop
022F 015C     498      ajmp      acc_0      ;do this again
0230 18       499      mov      a,temp_reg1
0231 AE0B     507      mov      r6,count_fw ;get no. steps past post from count-fw
0233 7400     508      mov      a,#decel_steps
0235 C3       509      clr      c
0236 9E       510      subb      a,r6
0237 FE00XX00 511      mov      r6,a ;R6 now has decel_steps - count_fw
0238 7B001000 513      mov      r3,#decel_decr ;decel decr is now in r3
023A E50F     514      mov      a,r_ptr ;check to see how far into accel tab
023C C3       515      clr      c
023D 9400     516      subb      a,#acc_steps ;C set if r_ptr < top_speed
023F 5002     517      jnc      dec_0a
0241 7B01     518      mov      r3,#1 ;this is the decel_decr value
0243 EE       520      dec_0a:      mov      a,r6
0244 8BF0     521      mov      b,r3

```

IEEE Micro Mouse using the 87C751 microcontroller AN443

MCS-51 MACRO ASSEMBLER 751MAIN

04/16/92 PAGE 11

```

LOC OBJ          LINE      SOURCE
0246 A4           522      mul     ab
0247 04           523      inc     a
0248 F50F         524      mov     r_ptr,a
024A F50E         525      mov     l_ptr,a
024C 300BFD       527      dec_0:  jnb     time_int,$ ;stay here until int happens
024F C20B         528      clr     time_int
0251 71B7         529      acall    snapshot ;take a look at the walls
0253 300C08       531      jnb     r_int,dec_1 ;skip if right hasn't been stepped yet
0256 C20C         532      clr     r_int
0258 E50F         533      mov     a,r_ptr ;get right pointer
025A C3           534      clr     c
025B 9B           535      subb    a,r3 ;slow down by getting lesser value
025C F50F         536      mov     r_ptr,a ;ptr just got decremented
025E 300DEB       539      dec_1:  jnb     l_int,dec_0 ;if need to do left then goto dec-1
0261 C20D         540      clr     l_int
0263 E50E         541      mov     a,l_ptr ;get left pointer
0265 C3           542      clr     c
0266 9B           543      subb    a,r3 ;slow down by getting lesser value
0267 F50E         544      mov     l_ptr,a ;ptr just got decremented
0269 DEE1         545      dec_2:  djnz    r6,dec_0 ;decr number of steps to decel
026B C28C         546      clr     tr0 ;stop the timer int
026D 7F00         547      mov     r7,#pause_val ;pause value
026F 9145         548      acall    pause ;stay stationary for a while
0271 E58B         557      mov     a,rtl
0273 C3           558      clr     c
0274 9400         559      subb    a,#pivot_speed ;slower speed
0276 30140D       560      jnb     make_180,turn90 ;do 180 else turn
0279 101706       561      jbc     cw180,piv_0 ;180 cw
027C D217         562      setb    cw180 ;next time 180 ccw
027E C293         563      clr     l_dir ;make L go backwards
0280 4190         564      ajmp    piv_1
0282 C291         565      piv_0:  clr     r_dir ;make R go bw
0284 4190         566      ajmp    piv_1
0286 A208         568      turn90:  mov     c,r_turn ;this section sets the motor dir
0288 B3           569      cpl     c ; bits according to which
0289 9291         570      mov     r_dir,c ; dir it has to turn.
028B A209         571      mov     c,l_turn
028D B3           572      cpl     c
028E 9293         573      mov     l_dir,c
0290 201404       576      piv_1:  ;THIS IS NEW. Mod adds or subtracts steps depending on sensor info
0290 201404       576      jb     make_180,piv_2y ;if mouse is skewed, it turns more

```

IEEE Micro Mouse using the 87C751 microcontroller AN443

MCS-51 MACRO ASSEMBLER 751MAIN

04/16/92 PAGE 12

```

LOC OBJ      LINE      SOURCE
0293 7F00      F      577      mov     r7,#half_90      ; or less degrees
0295 8002      578      sjmp     piv_2x
0297 7F00      F      579      piv_2y:  mov     r7,#half_180      ;
580      piv_2x:;this section determines add or subt count depending on dir to pivot
581      ;(r_dir & too_l) | (l_dir & too_r) = - steps
582      ;(r_dir & too_r) | (l_dir & too_l) = + steps
0299 A291      583      mov     c,r_dir      ;check for first condition
029B 8201      584      anl     c,too_l
029D 9222      585      mov     temp_bit1,c
029F A293      586      mov     c,l_dir
02A1 8200      587      anl     c,too_r
02A3 7222      588      orl     c,temp_bit1
02A5 5003      589      jnc     piv_2z
02A7 1F        590      dec     r7      ;decr step count to do pivot
02A8 800F      591      sjmp     piv_2v
02AA A291      592      piv_2z:  mov     c,r_dir      ;check for second condition
02AC 8200      593      anl     c,too_r
02AE 9222      594      mov     temp_bit1,c
02B0 A293      595      mov     c,l_dir
02B2 8201      596      anl     c,too_l
02B4 7222      597      orl     c,temp_bit1
02B6 5001      598      jnc     piv_2v
02B8 0F        599      inc     r7      ;incr step count to do pivot
600
02B9 750F00     601      piv_2v:  mov     r_ptr,#0      ;point at first accel value
02BC 750E00     602      mov     l_ptr,#0
02BF 750DFE     603      mov     r_timr,#0feh      ;even up the timers to step
02C2 750CFE     604      mov     l_timr,#0feh      ; evenly
02C5 D28C      605      setb    tr0      ;enable timer
606
02C7 300CFD     607      piv_2:   jnb     r_int,$      ;stay here until done
02CA C20C      608      clr     r_int      ;ack r interrupt
02CC 050F      609      inc     r_ptr      ;incr step count
02CE 050E      610      inc     l_ptr      ;incr step count
02D0 E50F      611      mov     a,r_ptr      ;see if at half way mark
02D2 201405     612      jb     make_180,piv_2a
02D5 B507EF     613      cjne    a,07,piv_2      ;IF half way fall thru to piv_3
02D8 8003      614      sjmp     piv_3
02DA B507EA     615      piv_2a:  cjne    a,07,piv_2      ;if half way, fall thru
616
02DD B40002     F      617      piv_3:   cjne    a,#look90,piv_4
02E0 D21B      618      piv_3a:  setb    look      ;if so, set look bit
02E2 300CFD     619      piv_4:   jnb     r_int,$      ;stay here until int done
02E5 C20C      620      clr     r_int      ;ack int
02E7 150F      621      dec     r_ptr      ;make it slow down
02E9 150E      622      dec     l_ptr
02EB 301B11     623      jnb     look,piv_5      ;if look is set, do snapshot
624
02EE 71B7      625      acall   snapshot      ;if aligned & wall exist truth table:
02F0 20051C     626      jb     aligned,piv_end
02F3 A201      627      mov     c,too_l
02F5 B093      628      anl     c,/l_dir
02F7 4016      629      jc     piv_end
02F9 A200      630      mov     c,too_r
02FB B091      631      anl     c,/r_dir

```


IEEE Micro Mouse using the 87C751 microcontroller AN443

MCS-51 MACRO ASSEMBLER 751MAIN

04/16/92 PAGE 13

```

LOC OBJ      LINE      SOURCE
02FD 4010     632      jc      piv_end
                                633
02FF E50F     634      mov     a,r_ptr      ;
0301 70DA     635      jnz     piv_3      ;IF end of count fall thru
                                636
0303 A202     637      mov     c,r_wall
0305 7203     638      orl     c,l_wall
0307 5006     639      jnc     piv_end      ;IF no walls get out
                                640
0309 050F     641      inc     r_ptr      ;keep steppin while you see
030B 050E     642      inc     l_ptr      ; a wall
030D 41E2     643      ajmp    piv_4
                                644
                                645      piv_end:
030F C28C     646      clr     tr0      ;off timer int
0311 850A8B    647      mov     r7,ls373      ;full speed again
0314 C21B     648      clr     look      ;
0316 C20C     649      clr     r_int
0318 C20D     650      clr     l_int
031A 43900A    651      orl     pl,#00001010b ;set r dir l dir
031D 7F00     652      mov     r7,#pause_val ;pause for a bit
031F 9145     653      acall   pause
                                654
0321 0141     655      ajmp    accel      ;goto accel
                                656
                                657
                                658      ;*****
0323 C28C     659      ; This is the Timer 0 interrupt subroutine.
                                660      ; It will step a motor if the "prescale" for the corresponding
0325 C0D0     661      ; motor overflows (or decrements to zero).
                                662      ;*****
0327 C0E0     663
0329 C082     664      tim_0:      ;used to step the motors
032B C083     665      clr     tr0      ;quit counting
032D 900000    666      push    psw
0330 D50D16    667      push    acc
0333 C292     668      push    dpl
0335 E50F     669      push    dph
0337 93       670      mov     dptr,#acc_tab
0338 F50D     671      djnz    r_timr,tim_00
033A E508     672      clr     r_step      ;step the R motor
033C B48202    673      mov     a,r_ptr      ;load timer value from pointer
033F 8002     674      movc    a,@a+dptr      ;get higher byte accel value
0341 0508     675      mov     r_timr,a      ;load the timer value from table
0343 D20C     676      mov     a,step_count
0345 D20B     677      cjne    a,#130,tim_x
0347 D292     678      sjmp    tim_y
0349 D50C0D    679      tim_x:      inc     step_count      ;decide uses this stuff
034B D20C     680      tim_y:      setb    r_int      ;flag that the R mot was stepped
034D D292     681      setb    time_int      ;int has occurred flag
034F D50C0D    682      setb    r_step
0351 D50C0D    683      tim_00:    djnz    l_timr,tim_ret ;get out if not zero
0353 C294     684      clr     l_step      ;step the L motor
0355 E50E     685      mov     a,l_ptr      ;load timer value from pointer
0357 93       686      movc    a,@a+dptr      ;get higher byte accel value

```

IEEE Micro Mouse using the 87C751 microcontroller AN443

MCS-51 MACRO ASSEMBLER 751MAIN

04/16/92 PAGE 14

```

LOC OBJ          LINE      SOURCE
0351 F50C         687          mov     l_timr,a      ;load the timer value from table
0353 D20D         688          setb    l_int       ;flag that the L mot was stepped
0355 D20B         689          setb    time_int    ;int has occurred flag
0357 D294         690          setb    l_step
                                691          tim_ret:
0359 D083         692          pop     dph
035B D082         693          pop     dpl
035D D0E0         694          pop     acc
035F D0D0         695          pop     psw
0361 D28C         696          setb    tr0           ;start counting
0363 32           697          reti
                                698
                                699
                                700          ;*****
0364 202010       701          ; This is the External interrupt 1 subroutine.
                                702          ;*****
                                703
                                704          int_1:      ;used for setting left or right hug
                                705          ; L/R=0 right algo, L/R=1 left algo
                                706          ;AND also setting speed of run
0366 10210B       707          jbc     done,int_1_2    ;incr speed and get out
036A D221         708          setb    l_r_bit      ;hug left
036C C282         709          clr     l_led
036E 7F0A         710          mov     r7,#10      ;value for 2 seconds
0370 9145         711          acall    pause
0372 D282         712          setb    l_led
0374 32           713          reti
                                714
0375 C221         715          int_1_0:    clr     l_r_bit      ;hug right
0377 7E05         716          int_1_2:    mov     r6,#5
0379 302002       717          int_1_1:    jnb     done,int_1_3
037C 058B         718          inc     rtl           ;incr the speed
037E C282         719          int_1_3:    clr     l_led
0380 7F02         720          mov     r7,#2
0382 9145         721          acall    pause
0384 D282         722          setb    l_led
0386 7F02         723          mov     r7,#2
0388 9145         724          acall    pause
038A DEED         725          djnz    r6,int_1_1
038C 32           726          reti
                                727
                                728          ;*****
038D C28C         729          ;*****
                                730          ; This is the External interrupt 0 subroutine.
                                731          ;*****
                                732
                                733          int_0:      ;used for starting and stopping the thing
038F 200A06       734          clr     tr0
0392 D20A         735          jb     s_s_int, int_0_0    ;we are here to start up
0394 C290         736          setb    s_s_int
0396 61AE         737          clr     mot_en
0398 D290         738          ajmp    int_0_ret
039A D220         739          int_0_0:    setb    mot_en      ;we are here cause at finish box
                                740          setb    done      ;tell the prog that we are done
                                741

```

IEEE Micro Mouse using the 87C751 microcontroller AN443

MCS-51 MACRO ASSEMBLER 751MAIN

04/16/92 PAGE 15

```

LOC OBJ      LINE      SOURCE
039C C20A     742      clr      s_s_int
039E 90001D   743      mov      dptr,#main_1 ;get address
03A1 A881     744      mov      r0,sp ;set up to start all over
03A3 A683     745      mov      @r0,dph ;load reti vector to 0004h
03A5 18       746      dec      r0
03A6 A682     747      mov      @r0,dpl
03A8 75A800   748      mov      ie,#00
03AB 758800   749      mov      tcon,#00 ;clears any ints
                                750
03AE C297     751      int_0_ret:  clr      r_led
03B0 7F14     752      mov      r7,#20 ;value for 2 seconds
03B2 9145     753      acall  pause
03B4 D297     754      setb   r_led
03B6 32       755      reti
                                756
                                757
                                758 ;*****
                                759 ; This section strobes the sensors for data. *
                                760 ;*****
                                761
                                762 snapshot: ;takes a look at walls and sets bits accordingly
                                763 ;R4, R5 for sensor info. ACC, C, r l sens, bit addressables
03B7 75B0FF   764      mov      p3,#0ffh
03BA 752000   765      mov      ss_bits,#0 ;clear all flags
03BD C281     766      clr      r_sens ;enable right sensor bank
03BF A4       767      mul      ab ;causes a 6uS wait state
03C0 A4       768      mul      ab
03C1 A4       769      mul      ab
03C2 A4       770      mul      ab
03C3 E5B0     771      mov      a,p3 ;store right wall
03C5 D281     772      setb   r_sens
03C7 FC       773      mov      r4,a ;wall is now repr as a high
03C8 33       774      rlc      a ;store R sens_0 for f_wall
03C9 9204     775      mov      f_wall,c
03CB 6002     776      jz      ss_0 ;if no wall goto ss_0
03CD D202     777      setb   r_wall ;right wall present
                                778
03CF C280     779      ss_0:  clr      l_sens ;enable left sensor bank
03D1 A4       780      mul      ab ;causes a 6uS wait state
03D2 A4       781      mul      ab
03D3 A4       782      mul      ab
03D4 A4       783      mul      ab
03D5 E5B0     784      mov      a,p3 ;store left wall
03D7 D280     785      setb   l_sens
03D9 FD       786      mov      r5,a ;wall is now repr as a high
03DA 33       787      rlc      a ;grab inner sens and or it
03DB 7204     788      orl      c,f_wall
03DD 9204     789      mov      f_wall,c ;store front wall
03DE 6002     790      jz      ss_2 ;if no wall goto ss_2
03E1 D203     791      setb   l_wall ;left wall present
                                792
03E3 20045E   793      ss_2:  jb      f_wall,ss_ret ;if no front wall then cont
03E6 7400     794      mov      a,#0 ;build case statement
03E8 A221     795      mov      c,l_r_bit ;
03EA 33       796      rlc      a ;00000| L/R | L | R |

```

IEEE Micro Mouse using the 87C751 microcontroller AN443

MCS-51 MACRO ASSEMBLER 751MAIN

04/16/92 PAGE 16

```

LOC  OBJ          LINE    SOURCE
03EB  A203          797      mov     c,l_wall      ; |algo |wall|wall|
03ED  33            798      rlc     a              ; | | |bit |bit |
03EE  A202          799      mov     c,r_wall      ; | | | | |
03F0  33            800      rlc     a              ;done shifting in bits for case stmnt
03F1  B40102        801      cjne    a,#01,ss_4      ;chk R
03F4  810F          802      ajmp    ss_9
03F6  B40302        803      cjne    a,#03,ss_5      ;chk R
03F9  810F          804      ajmp    ss_9
03FB  B40502        805      cjne    a,#05,ss_6      ;chk R
03FE  810F          806      ajmp    ss_9
0400  B40202        807      cjne    a,#02,ss_7      ;chk L
0403  8115          808      ajmp    ss_10
0405  B40602        809      cjne    a,#06,ss_8      ;chk L
0408  8115          810      ajmp    ss_10
040A  B40737        811      cjne    a,#07,ss_ret      ;if eq then chk L else do nothing
040D  8115          812      ajmp    ss_10
                        813      ;*****
                        814      ss_9:      ;check the right side offset
040F  8CF0          815      mov     b,r4          ;put wall info into acc
0411  C21E          816      clr     genp2          ;0=chk R
0413  8004          817      sjmp    ss_93
0415  8DF0          818      mov     b,r5
0417  D21E          819      setb    genp2          ;1=chk L
                        820
0419  E5F0          821      ss_93:      mov     a,b
041B  5400          822      anl     a,#sens_pat      ;masking for cmp, 3 high 4 low
041D  B40804        823      cjne    a,#08h,ss_90      ;check for aligned condition
0420  D205          824      setb    aligned          ;perfectly on wall
0422  8144          825      ajmp    ss_ret
0424  E5F0          826      ss_90:      mov     a,b          ;get wall info again
0426  5403          827      anl     a,#00000011b      ;check for too close center if a > 0
0428  600B          828      jz     ss_91          ;if no wall on ones then not too
042A  201E04        829      jb     genp2,ss_101
042D  D201          830      setb    too_l
042F  8144          831      ajmp    ss_ret
0431  D200          832      ss_101:      setb    too_r
0433  8144          833      ajmp    ss_ret
0435  E5F0          834      ss_91:      mov     a,b          ;get wall info again
0437  5460          835      anl     a,#01100000b      ;check for too close wall if a > 0
0439  6009          836      jz     ss_ret
043B  201E04        837      jb     genp2,ss_102
043E  D200          838      setb    too_r
0440  8144          839      ajmp    ss_ret
0442  D201          840      ss_102:      setb    too_l
0444  22            841      ss_ret:      ret
                        842
                        843      ;*****
                        844      pause:      ;pause loop using R7 as the loop counter
0445  C28C          845      clr     tr0
0447  903EFE        846      mov     dptr,#03efeh
044A  D582FD        847      djnz    dpl,$
044D  D583FA        848      djnz    dph,$-3
0450  DFF3          849      djnz    r7,pause
0452  22            850      ret
                        851

```

IEEE Micro Mouse using the 87C751 microcontroller AN443

MCS-51 MACRO ASSEMBLER 751MAIN

LOC OBJ LINE SOURCE

```

852 ;*****
853 halt: ;This S.R. brings the thing to a halt -- mostly used for debug
854 clr et0
855 setb mot_en
856 sjmp $+1
857 ;*****
858 EXTRN CODE (acc_tab) ;from the DOS file 751acc.asm
859
860

```

$$\text{Table entry} = \frac{\text{Qoc/MHz}}{\text{Band Rate}} \times \frac{2}{15}$$

Remember that the table entry is a two byte value, so the result of the above must be split into upper and lower bytes (we have a hexadecimal calculator, it may also be possible to get the assembler to do all the calculations for you).

The above equation was derived as follows:

$$\text{maximum timer value} = \frac{\text{machine cycle time}}{\text{table entry}}$$

$$\text{recognition time} = \frac{\text{table entry}}{\text{machine cycle time}} \times \text{byte time}$$

Note: 8-bit data, the number of "valid" bits is 8, and data-to-recognition (the minimum 8 bits to recognition) is 8 for 8-11 communication.

$$\text{byte time} = \frac{1}{\text{clock rate}} \times 8 \text{ bits}$$

$$\text{machine Qoc frequency} = \frac{15}{\text{byte time}}$$

There is an assumption in this method that anyone using it needs to be aware of. That is, this technique depends on only one character being received during the sampling window which has to be at least as long as a typical character at the slowest baud rate that can be accepted. Essentially this means that the data must normally come from someone typing at a keyboard.

On our PC, we were not able to fool the program by typing two characters in quick succession. The PC function keys did present a problem because they send two characters in a tight sequence, and fooled the program into detecting the wrong baud rate. In the example program, which is designed for a 15 MHz clock, the total sample interval is about 60 milliseconds, or about twice the duration of an RS-232 character sent at 300 baud.

It partly is used, a possibility of a baud rate determination error happens when the four 8255 and the parity bit of the character received are all ones. This can happen for the lower case letters "n", "through", "r", plus only brackets, without the "(", ")", and "quotes", depending on whether the system uses odd or even parity. Note that the usual prompt characters that a user would type to get a system's attention (e.g., "quit", "continue", "return", and "escape") are NOT subject to this limitation.

04/16/92 PAGE 17

This note is a continuation of the previous note. It should be read in conjunction with the previous note.

This can eliminate the need to have setup routines which are difficult to write.

This can eliminate the need to have setup routines which are difficult to write.

This can eliminate the need to have setup routines which are difficult to write.

This can eliminate the need to have setup routines which are difficult to write.

This can eliminate the need to have setup routines which are difficult to write.

Automatic baud rate detection for the 80C51 AN447

Author: Greg Goodhue

This note documents a method to automatically establish the correct baud rate for serial communications in many 80C51 family applications. The first character received after a program is started is used to measure the baud rate empirically.

This can eliminate the need to have setup switches whose settings are difficult to remember and all of the other headaches associated with applications that use multiple baud rates. One might assume that a reliable method of accomplishing this might be impossible without severely limiting the characters that could be recognized. The problem is in finding a timing interval that can be measured in a large number of possible characters under a wide variety of conditions.

Measuring a single bit time would be the obvious way to quickly determine what baud rate is being received. However, many ASCII characters don't have an example of a single bit time in the RS-232 pattern. For most characters, the length of the entire transmission from the start bit to the last "visible" transition will fall within certain ranges as long as some reasonable assumptions can be made about the possible baud rates (i.e. that they are standard baud rates). Moreover, many systems now use 8 data bits and no parity for ASCII transmissions. In this format, normal ASCII characters will never have the MSB set and since UARTs send data LSB first/MSB last, the program would always be able to "see" the beginning of the stop bit.

The following baud rate detection routine waits for a start bit (falling edge) on the serial input pin and then starts timer 0. At every subsequent rising edge of the serial data, the timer value is captured and saved. When the

timer overflows, the last captured value will indicate the duration of the serial character from the start bit to the last 0 to 1 transition (hopefully the stop bit).

The table CmpTable contains the maximum timer measurement that is accepted for each baud rate. These values were picked such that a timed interval of only 4 data bit times (plus the start bit time) will still produce the correct baud rate.

There is an assumption in this method that anyone using it needs to be aware of. That is, that this technique depends on only one character being received during the sampling window, which has to be at least as long as a typical character at the slowest baud rate that can be accepted. Essentially this means that the data must normally come from someone typing at a keyboard.

On our PCs, we were not able to fool the program by typing two characters in quick succession. The PC function keys did present a problem because they send two characters in a tight sequence, and fooled the program into detecting the wrong baud rate. In the example program, which is designed for a 12 MHz clock, the total sample interval is about 65 milliseconds, or about twice the duration of an RS-232 character sent at 300 baud.

If parity is used, a possibility of a baud rate determination error happens when the four MSBs and the parity bit of the character received are all ones. This can happen for the lower case letters "p" through "z", plus curly brackets, vertical bar (|), tilde (~), and "delete", depending on whether the system uses odd or even parity. Note that the usual prompt characters that a user would type to get a system's attention (e.g. space, carriage return, and escape) are NOT subject to this limitation.

Because of the way this program works, the first input character that is used to detect the baud rate is lost since the UART cannot be set to the correct baud rate until after the first character has been timed. Also, most "real" programs using this technique would want to repeat the baud rate detection process if framing errors are detected at the UART during normal operation.

To calculate CmpTable values for other oscillator frequencies and baud rates, use the following equation:

$$\text{Table entry} = \frac{\text{Osc(MHz)}}{\text{Baud Rate}} \times \frac{5}{12}$$

Remember that the table entry is a two byte value, so the result of the above must be split into upper and lower bytes (easy if you have a hexadecimal calculator). It may also be possible to get the assembler to do all of the calculations for you.

The above equation was derived as follows:

$$\frac{\text{maximum timer value}}{\text{(table entry)}} = \frac{\text{minimum recognition time}}{\text{machine cycle time}}$$

$$\text{Minimum recognition time} = \frac{\text{bits-to-recognize}}{\text{\#-of-bits}} \times \text{byte time}$$

Note: '#-of-bits' (the number of "visible" bits) is 9, and bits-to-recognize (the minimum # of bits to recognize) is 5 for 8-N-1 communication.

$$\text{byte time} = \frac{1}{\text{baud rate}} \times \text{\#-of-bits}$$

$$\text{machine cycle time} = \frac{\text{Osc frequency}}{12}$$

Automatic baud rate detection for the 80C51 AN447

```

;*****
;
; Automatic Baud Rate Detection Test
;*****

$Title(Automatic Baud Rate Detection Test)
$Date(12-16-91)
$MOD552

;*****
; Definitions
;*****

RX      BIT    P3.0      ;Location of serial receive pin.
CharH   DATA  30h      ;Holds high byte of frame timer result.
CharL   DATA  31h      ;Holds low byte of frame timer result.
BaudRate DATA  32h      ;Holds final baud rate determination.
Display EQU    P4        ;Port to display result for debug.

;*****
; Reset and Interrupt Vectors
;*****

ORG 8000h

Start:  ACALL AutoBaud    ;Go try to get a baud rate value.
        MOV  Display,BaudRate ;Display baud rate value for debug.
        SJMP Start

;*****
; Subroutines
;*****

; AutoBaud Rate Detect Routine.
; Attempts to detect baud rate from first received character, by measuring
; the length of the character. Some characters may not work properly,
; primarily those that end with more than 3 (4?) ones in a row.
; Returns with ACC = baud rate pointer.

AutoBaud: MOV  TMOD,#01h    ;Initialize timer 0 (UART baud rate timer).
           MOV  TH0,#0      ;Put timer 0 in 16-bit counter mode.
           MOV  TL0,#0
           MOV  TCON,#0

           MOV  CharH,#0    ;Initialize timer result.
           MOV  CharL,#0

AB0:      JB   RX,AB0       ;Wait for serial start bit.
           SETB TR0         ;Start timer.

AB1:      JB   TF0,AB3      ;Check for timer overflow.
           JNB  RX,AB1      ;Check for a rising edge on serial data.
           MOV  CharH,TH0    ;Capture timer value at this serial edge.
           MOV  CharL,TL0

AB2:      JB   TF0,AB3      ;Check for timer overflow.
           JB   RX,AB2      ;Check for falling edge on serial data.

```

Automatic baud rate detection for the 80C51 AN447

```

                SJMP AB1                ;Go back and repeat sampling.
AB3:            CLR TR0                ;Maximum sample time has expired, check result.
                CLR TF0                ;Begin by stopping timer and clearing flag.

                MOV BaudRate,#19        ;Set up table pointers.
CmpLoop:        MOV A,BaudRate
                MOV DPTR,#CmpTable
                MOVC A,@A+DPTR          ;Get a table entry for comparison.
                DEC BaudRate
                CJNE A,CharH,Cmp1       ;Check result range.
                SJMP CmpLow             ;High byte table = timed value, check low byte.
Cmp1:           JC CmpMatch            ;A match if table value is < timed value.
                DJNZ BaudRate,CmpLoop   ;Check for end of comparison table.
                SJMP CmpMatch

CmpLow:         MOV A,BaudRate
                MOVC A,@A+DPTR          ;Get a table entry for comparison.
                CJNE A,CharL,Cmp2       ;Check result range.
                SETB C                  ;Match if equal.
Cmp2:           JC CmpMatch            ;C set if A < low byte of result.
                DJNZ BaudRate,CmpLoop   ;Check for end of comparison table.

CmpMatch:       MOV A,BaudRate          ;Comparison complete,
                CLR C                  ; get final baud rate index,
                RRC A
                MOV BaudRate,A          ; and save.
                RET

```

; Compare table holds timer values for the transition points of the accepted
; baud rates. Entries are LSB, MSB. These values are for 12 MHz operation.

```

CmpTable:
DB 40h,0        ;0 - out of range, value too low.
DB 80h,0        ;1 - 38400 baud.
DB 0,01h        ;2 - 19200 baud.
DB 0,02h        ;3 - 9600 baud.
DB 0,04h        ;4 - 4800 baud.
DB 0,08h        ;5 - 2400 baud.
DB 0,10h        ;6 - 1200 baud.
DB 0,20h        ;7 - 600 baud.
DB 0,40h        ;8 - 300 baud.
DB 0,80h        ;9 - out of range, value too high.

END

```

Determining baud rates for 8051 UARTs and other UART issues

AN448

Author: Greg Goodhue

The purpose of this note is to expand upon and clarify some aspects of determining baud rates and crystal frequencies for using a standard 8051 or 80C51 UART for ordinary RS-232 type serial communication. The standard baud rate equation is simplified here and is restated to allow solving for other variables such as the crystal frequency and timer reload values.

The following discussion assumes that the reader has some knowledge of the

8051/80C51 UART and timers. This should be considered a supplement to the information presented in the Philips 80C51 Family Microcontroller Data Book sections on Timer/Counters and the Standard Serial Interface.

Since this discussion assumes the use of a standard UART for RS-232 serial communications, the UART will be used in modes 1 or 3 (variable baud rates) and timer

1 will be used in mode 2 (8-bit auto-reload mode) as the baud rate generator. All of the equations shown here give an option for two clock divisors depending on whether the SMOD bit is used on a CMOS microcontroller. For an NMOS device, always use the default value (SMOD is not = 1).

The basic equation for a timer reload value can be stated as:

$$TH1 = 256 - \frac{\text{Crystal Frequency}/384 \text{ (or 192 if SMOD = 1)}}{\text{Baud Rate}}$$

Example:

To obtain a timer reload value for a 9600 baud serial data rate with an 11.0592 MHz crystal:

$$256 - \frac{11,059,200/384}{9600} = 256 - 3 = 253, \text{ or FD hexadecimal}$$

The equation can also be solved to derive the baud rate or the crystal frequency from the other information as follows:

$$\text{Baud Rate} = \frac{\text{Crystal Frequency}/384 \text{ (or 192 if SMOD = 1)}}{256 - TH1}$$

$$\text{Minimum crystal frequency for a given baud rate} = \text{Baud Rate} \times 384 \text{ (or 192 if SMOD = 1)}$$

Thus, the minimum crystal frequency that may be used for 19.2k baud communication on a CMOS part with SMOD = 1 would be 19200×192 , which gives 3.6864 MHz. When using this equation, the timer reload value TH1 for the maximum baud rate is always 255 (256 - 1) or FF hexadecimal.

Of course, any even multiple of the frequency obtained in this manner will also support the same baud rate with a different timer reload value. For instance, four times 3.6864 MHz is 14.7456 MHz. At that crystal frequency, 19.2k baud is attained with a timer reload value that gives one fourth of the timer overflow rate: 252 (256 - 4) or FC hexadecimal.

CRYSTAL FREQUENCIES USED FOR STANDARD BAUD RATES

The following chart shows possible crystal frequencies for use with the 80C51 UART at standard baud rates. The chart assumes use of the UART in modes 1 or 3 (variable baud rates) and timer mode 2 (8-bit auto-reload mode). The chart also assumes a minimum requirement of at least 9600 baud (including the use of SMOD for baud rate doubling). More crystal frequencies are available if a lower maximum baud rate is required.

The minimum timer count column indicates how many timer counts are required at the

stated crystal frequency in order to obtain the maximum baud rate shown. The last column shows the timer reload value that is used to obtain the minimum timer count. This is simply 256 minus the minimum timer count.

Timer reload values for other baud rates at the same crystal frequency are determined by multiplying the minimum timer count by two successively and calculating a new reload value as previously mentioned. For instance, for 4800 baud at 1.8432 MHz, the timer count would be 2 (twice what it is for 9600 baud), giving a timer reload value of 254 (256 - 2) or FE hexadecimal.

Determining baud rates for 8051 UARTs and other UART issues

AN448

Maximum Standard Crystal (MHz)	Maximum Baud Rate	Timer Count	Timer Reload Value (in hex)
1.8432	9600	1	FF
3.6864	19200	1	FF
5.5296	9600	3	FD
7.3728	38400	1	FF
9.2160	9600	5	FB
11.0592	19200	3	FD
12.9024	9600	7	F9
14.7456	76800 (2 × 38400)	1	FF
16.5888	9600	9	F7
18.4320	19200	5	FB
20.2752	9600	11	F5
22.1184	38400	3	FD
23.9616	9600	13	F3
25.8048	19200	7	F9
27.6480	9600	15	F1
29.4912	153600 (4 × 38400)	1	FF
31.3344	9600	17	EF
33.1776	19200	9	F7
35.0208	9600	19	ED
36.8640	38400	5	FB

THE EFFECT OF USING OFF-FREQUENCY CRYSTALS

Occasionally, one may wish to use an off-frequency crystal in a design, but still want to make use of the UART for debug purposes.

Since most terminals (or other RS-232 devices) will communicate with another device that has a baud rate that is off by several percent, this can often be done successfully. WARNING: running the UART off-frequency is NOT recommended if part of the application's normal operation involves communication with other RS-232 devices.

There is no exact limit on how much frequency error is tolerable, since this depends on the devices communicating, the baud rates, precise frequencies used by both devices, etc. However, a rule of thumb may be used that the communication is likely to work if the frequency is off by less than 5%. This somewhat arbitrary number was arrived at as follows: for a ten-bit serial code (one start, 8 data bits, one stop), a 10% data rate error will put the receiver off by about plus or minus one bit time at the end of one data frame. A one bit-time error seems rather

excessive if one wants fairly reliable communications. So, consider using half of that value (5%) as a rule of thumb.

The consequence of all this is that one may often find a more standard "off-the-shelf" crystal frequency to use in an application, if the UART is being used for debugging, factory testing, etc. As an example, consider the well-known "color burst" crystal. At 3.579545 MHz, this crystal is only about 3% slower than the 3.6864 MHz crystal that may be desired for baud rate generation. As such, this lower cost crystal could be used in place of the less standard one in some cases. Another obvious replacement is to use the standard 14.31818 MHz crystal in place of the not-so-standard 14.7456 MHz crystal that appears in the table. This replacement also yields a less than 3% error and may be handy because it gives a fast instruction execution rate for the 80C51, whereas 3.58 MHz may be too slow for many applications.

It should also be remembered that RS-232 communications are most robust if characters are not transmitted back-to-back. This can become more important when the UART is deliberately used out of spec as described

here. When data is sent at full speed, there is no chance for the receiver to re-synchronize to the transmitted frames if it once gets out of synch. However, when there is a short pause between characters (about 2 to 3 bit times or longer), the receiver will generally be able to correctly locate start bits without framing errors. In the worst case, a pause of one byte-time or longer in a transmission should ALWAYS re-synchronize any receiver no matter how out of synch it has become.

A LITTLE KNOWN PHENOMENON

In the UART setup code for most applications, the actual timer count register (TL1) is not initialized. In many applications, this DOES have an effect on the way the UART behaves on the first character sent, although the chances of this being noticed are slight. This can be seen by trying to observe the first character sent from the UART on a logic analyzer that is being triggered by the end of microcontroller reset. The first character will begin so far down the time line that it will not be seen at any resolution on the logic analyzer that would show any of the individual bits.

Determining baud rates for 8051 UARTs and other UART issues

AN448

This effect occurs because TL1 has to time-out once before the first character is transmitted. If TL1 is not initialized in the program, it will have a reset value of 0. This could give the first timeout a duration of up to 255 normal bit times, depending on the reload value for TH1 (which again depends on the baud rate and crystal frequency).

Again, in most applications, this would never be an issue. In fact, it may often be an advantage to have a delay before the first serial character is sent after power-up. But if the first serial character should start sooner, TL1 may be initialized to some value other than zero. For no delay, the same value used in TH1 should be used.

Electro magnetic compatibility and printed circuit board (PCB) constraints

ESG89001

1. Introduction

The routing of the traces on a Printed Circuit Board (PCB) largely effect the ElectroMagnetic Compatibility (EMC) performance of the PCB with respect to both ElectroMagnetic (EM) radiation as susceptibility to EM-fields.

The PCB will connect electronic components such as passive components, transistors and ICs. Furthermore, cables to interconnect the PCB with other system parts e.g. another PCB, signalgenerator, CATV wall-outlet, DC-powersource or an AC-mains connection will largely influence the PCB with respect to EMC, [7].

In order to get a PCB on which the circuits function properly, the trace routing, the placement of components/ connectors and the decoupling used with certain ICs will have to be optimised according to the constrains given in this report.

To reach an economic and functional PCB design, the following items have to be beard in mind:

1. Correct choice of the PCB-format (mono, bi- or multi-layer),
2. Take care that "every" signaltrace has its signalreturn nearby,
3. Proper decoupling for each IC or group of ICs,
4. Allowed tracelengths and allowed loopareas,
5. Placement of the connectors,
6. Right cable choice with a proper connector,
7. Proper use and placement of filters and filterparts.

These items with the appropriate measures will be further explained.

THE MAIN TARGET IS TO GET CONTROL OVER YOUR PCB-CURRENTS.

Electro magnetic compatibility and printed circuit board (PCB) constraints

ESG89001

2. General

2.1. Conductors

Single conductors have, as a rule of thumb, an inductance of $1 \mu\text{H}/\text{m}$. At low frequencies only, below 1 kHz , R_{dc} applies. These impedances, together with the currents that will flow through these impedances, will be responsible for the voltage drop between points as Ohms law applies. The voltage drop can be diminished by either reducing the impedance or lowering the current through that impedance.

In typical digital designs the voltage drop will be frequency independent. A square wave current, resulting from a square wave output voltage to a resistive load, can be described as a series of sinewaves of which the amplitude of the harmonics decrease proportional with the frequency (Fourier expansions), see figure 1b. The impedance of the inductor increases proportional with frequency, see figure 1a, therefore the product; voltage drop, figure 1c, remains constant.

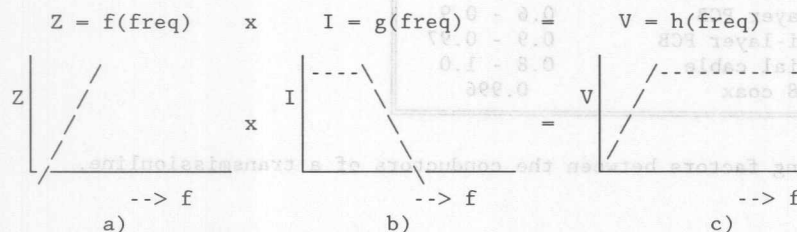


Figure 1. The relation between voltage drop as a result of current and impedance as function of frequency.

When the current has a triangular waveshape, as function of time, due to capacitive loading, the amplitude of the harmonics decreases with the frequency square and the voltage drop across the inductor reduces proportional with frequency.

2.2. Transmissionlines

By using the inductance of a single wire, L_1 , the mutual coupling, M , and the capacitance between the traces, C_1 , a transmissionline, shown in fig. 2, can be defined of which the characteristic impedance, Z_0 , equals:

$$Z_0 = \sqrt{(L_{\text{eff}} / C)}$$

where: $L_{\text{eff}} = L_1 + L_2 - 2 \cdot M$, $k = \sqrt{(L_1 + L_2) / M}$ and $C = C_1 + C_2$.

When the coupling, k , between the traces of the transmissionline is high, the effective inductance will decrease rapidly. Some coupling factors are given in table 1.

Electro magnetic compatibility and printed circuit board (PCB) constraints

ESG89001

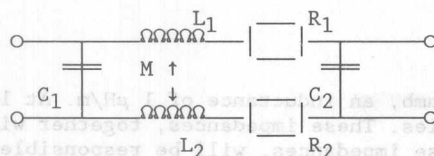
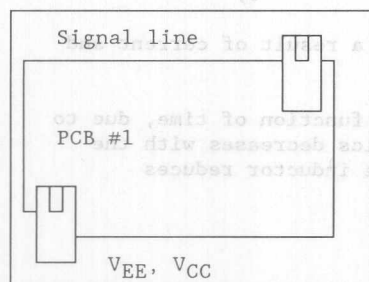


Figure 2. A segment of a transmission line and its network elements.

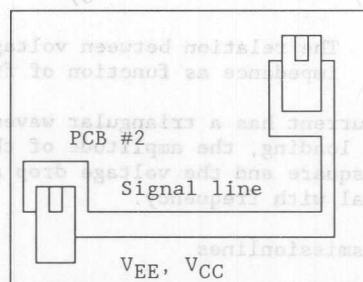
An indifferent signal path design, fig. 3a, can be changed into a transmission line design, fig. 3b. This change will lower the effective inductance, L_{eff} , between the two circuit blocks and will therefore lower the voltage drop between the two references of those circuits.

Transmissionline type	Coupling
parallel wires	0.5 - 0.7
bi-layer PCB	0.6 - 0.9
multi-layer PCB	0.9 - 0.97
coaxial cable	0.8 - 1.0
RG-58 coax	0.996

Table 1. Coupling factors between the conductors of a transmission line.



a) indifferent signal path
NO coupling between $S \cdot V_{EE}, V_{CC}$



b) Transmission line signal path
GOOD coupling between $S \cdot V_{EE}, V_{CC}$

Figure 3. Typical signal path design on a PCB.

2.3. Capacitive and inductive coupling

Separately, the capacitive and inductive values, derived from the definition of the transmission line, can also be used to calculate the crosstalk between adjacent traces, not being a function signal path. The capacitive coupling, representing an induced current, is given by:

Electro magnetic compatibility and printed circuit board (PCB) constraints

ESG89001

$$I_{Ck} = 1/C_k \cdot dV/dt,$$

where: C_k = coupling capacitance between adjacent traces; in practice: 100 pF/m. (depends upon the vicinity of other traces, see appendix A),

and the inductive coupling, representing an induced voltage, is given by:

$$V_{Mk} = M_k \cdot dI/dt,$$

where: M_k = mutual coupling between two traces, for further detail see chapter 4

In both coupling modes, the transfer function will typically show a high pass behaviour.

3. Choice of the PCB-material.

By a proper choice of the PCB-material and the routing of the traces a good transmissionline with low coupling to other traces can be created. Low coupling, or little crosstalk, can be obtained when the distance, d , between the transmissionline conductors is less than their distance to other adjacent conductors.

single layer:

$$d(S1*GND) < d(S2*S1)$$

bi-layer:

$$c) \quad d(S1*GND) < d(S2*S2) \quad \text{or} \quad d(S1*GND) \text{ AND } d(S2*GND) < d(S1*S2)$$

$$d) \quad d(S1*GND) \text{ AND } d(S2*GND) < d(S1*S2)$$

multi-layer:

$$d(Si*V_{EE}) \text{ OR } d(Si*V_{CC}) < d(Si*Sj) \\ 1 \leq i, j \leq \text{number of traces}$$

Figure 4. Typical applications of the PCB-format.

By using these examples of geometry of traces the definition of the transmissionline between $S1, S2, Si, j$ and $(S2) GND, V_{EE}$ and/or V_{CC} are well defined and the coupling between the traces $S2$ and $S1$ is low.

Electro magnetic compatibility and printed circuit board (PCB) constraints

ESG89001

The most economic PCB format has to be chosen based on:

- * the legal and/or functional EMC-requirements for the product,
- * trace density,
- * assembly and manufacturer capabilities,
- * CAD-system capabilities,
- * design-costs
- * PCB-quantities and
- * the costs of EM-shielding.

Special attention must be given to the integral costs (components packaging /pinning + PCB-format + EM-shielding + construction + assembly) when a product definition is considered by using a NON-shielded cover. In many cases the choice of a proper PCB-format may expel the need for a metallized box within the plastic cover.

To improve immunity and to lower unwanted emission, both in fast analog and all digital applications, transmissionlines are needed. Dependent upon the transition of the outputsignal a transmissionline needs to be present between $S \leftrightarrow V_{CC}$, $S \leftrightarrow V_{EE}$ and $V_{EE} \leftrightarrow V_{CC}$, as indicated in figure 5.

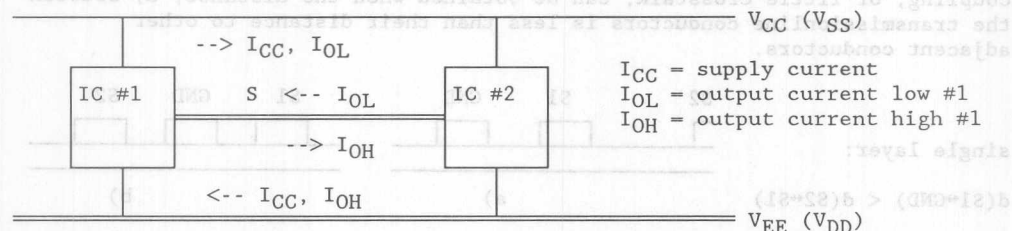


Figure 5. Typical diagram of an interconnection between (digital) ICs which shows 3 specific transmissionlines.

The signal current will be determined by the output-stage symmetry of the circuit. For MOS: $I_{OL} = I_{OH}$, while for TTL: $I_{OL} > I_{OH}$.

The Logic Family and functional reasons determine the typical characteristic impedance, Z_0 , for that transmissionline which is given in table 2.

Function/logic	Z_0 [Ω]
supply (typ)	$\ll 10$
signal ECL	50
signal TTL	100
signal HC(T)	200

Table 2. The transmissionline impedances, Z_0 , for several signal paths.

Electro magnetic compatibility and printed circuit board (PCB) constraints

ESG89001

For two traces next to each other the following formula applies [10,11].

$$Z_0 = \frac{120 \ln (\pi \cdot h / (b+c))}{\sqrt{\epsilon_r}}$$

where: h = distance between traces
 b = width of the trace
 c = thickness of the trace; typical 17 μm ,

for two traces on top of each other:

$$Z_0 = \frac{120 \pi (h / (h+b))}{\sqrt{\epsilon_r}}$$

where: h = 1.5 mm (typical thickness of epoxy).

When the trace is above a groundplane the following formula applies:

$$Z_0 = \frac{87 \ln (6 \cdot h / (.8 \cdot b + c))}{\sqrt{(\epsilon_r + \sqrt{2})}}$$

and in case of a trace between two (ground-)planes the formula yields:

$$Z_0 = \frac{60 \ln (4 \cdot K / (.67 \cdot \pi \cdot b \cdot (.8 + c/b)))}{\sqrt{\epsilon_r}}$$

where: K = distance in-between the planes

Typically the permittivity for epoxy material equals: $\epsilon_r = 4.7$

4. The signaltrace and its signalreturn.

Signaltraces need to have their signal-return-traces as close as possible in order to prevent emission from that looparea enclosed by these traces and to reduce susceptibility due to voltages which can be induced in this loop e.g. by RF-transmitters and ESD.

Commonly, when the distance between two traces equals the width of the traces, the coupling factor is about 0.5 to 0.6. The effective inductance of the traces has gone down from 1 $\mu\text{H}/\text{m}$ to 0.4 - 0.5 $\mu\text{H}/\text{m}$.

This means that 40 to 50 % of the signal-return current may run freely through the other traces of the PCB.

For each signal path between two (sub-)blocks either analog or digital three properly defined transmissionlines need to be present with the impedances given in table 2 and shown in figure 5.

With TTL logic the sink-current; the high-to-low transition, is higher than the source-current. In this case the transmissionline should be defined between V_{CC} and S instead of V_{EE} and S, which is commonly considered.

The mutual coupling between two parallel traces can be calculated from the double integral [9]:

Electro magnetic compatibility and printed circuit board (PCB) constraints

ESG89001

$$M_k = \mu / (4 \cdot \pi) \cdot \int_{l_1} \int_{l_2} ds_1 \cdot ds_2 \cdot dr / |r|$$

where: l_1, l_2 = length of traces 1 and 2
 r = relative distance between line segments, ds_1, ds_2 , of each trace.

Substituting the geometry of two parallel lines results in:

$$M_k = 200 [1.1 \ln \{ (1 + \sqrt{(l^2 + h^2)}) / h \} + \sqrt{(l^2 + h^2)} + h] [nH]$$

where: l = length of the two parallel traces and
 h = distance between the traces (trace thickness and width are neglected).

If the coupling between the two conductors of a transmissionline is too low, a ferrite toroid ($\mu_r > 200$ (-5000)), with some windings, will increase this coupling to ≈ 1 .

By using ferrite toroids one can get full control over the signal- and signal-return currents.

In case of parallel conductors, the characteristic impedance of this transmissionline may be influenced by the ferrite. In case of coaxial cable, the presence of the ferrite will only be noticeable on the outer parameters of the cable.

CONCLUSIONS:

- I. Use traces as thin as possible next to one another instead of on top of each other (separation commonly less than 1.5 mm ÷ epoxy thickness of a bi-layer).
- II. Create a lay-out where every signalline has his signal-return at the closest possible interval (applies to both signal- and supply-traces).
- III. If the coupling between the conductors of the transmission-line is insufficient a ferrite toroid may be used.

5. Proper decoupling with each IC.

ICs will be commonly decoupled by capacitors only. Because capacitors are not ideal, resonances will occur. Above the resonance frequency the capacitor behaves as an inductor which means that the dI/dt is limited. The value of this capacitor is determined by the voltage-fluctuations which are allowed across the powersupply pins of the IC. According to good designers practice, this voltage fluctuation should be less than 25 % of the signal-line worst-case noise margin. From the following equation the optimal decoupling capacitor for each logic family output gate can be calculated:

$$I = C \cdot dV/dt$$

Electro magnetic compatibility and printed circuit board (PCB) constraints

ESG89001

The worst-case signal-line noise margins for a several logic families are given in table 3, together with the recommended decoupling capacitor value, $C_{dec.}$, which need to be added with each output gate.

FAMILY	NOISE-MARGIN volt	dI mA	/ dt ns	$C_{dec.}$ nF
CMOS(5V)	1.75	2	100	0.5
TTL-LS	.4	50	10	5.0
TTL-F	.4	50	2-3	22.0
HCT	.7	50	2-3	12.8
HC(5V)	1.2	50	2-3	7.5
ACT	1.7	175	1-2	35.0

Table 3. Recommended decoupling capacitor.

The values of the decoupling capacitors for fast logic families may no longer be useful if the capacitor incorporates a large series inductance, either caused by the construction of the capacitor, long connecting wires or PCB traces. Additional small ceramic capacitors (100-1000 pF) need then to be added, as close as possible to the pins of the IC, in parallel to these "LF-"decoupling capacitors. The resonance frequency of this ceramic capacitor (including the trace length towards the supply pins of the IC) should be above the bandwidth of the logic $[1/(\pi \cdot \tau_r)]$, where τ_r is the voltage risetime of the logic.

If the decoupling capacitor is placed with every IC the signalreturn current may chose which path is most convenient, V_{EE} or V_{CC} . This choice is determined by the mutual coupling present between the signaltrace and one of the supply traces.

Between two decoupling capacitors, one for each IC, and the inductance, L_{trace} , formed by the supply traces, a series resonant circuit will result. This resonance is only allowed when it occurs at low frequencies (< 1 MHz) or when the Q of this resonance circuit is low (< 2). This resonance can be kept below 1 MHz by using a choke with high RF-losses in series with the V_{CC} network and the decoupled IC. Too less RF-losses can be compensated by either adding a resistor in parallel or in series, fig. 6.

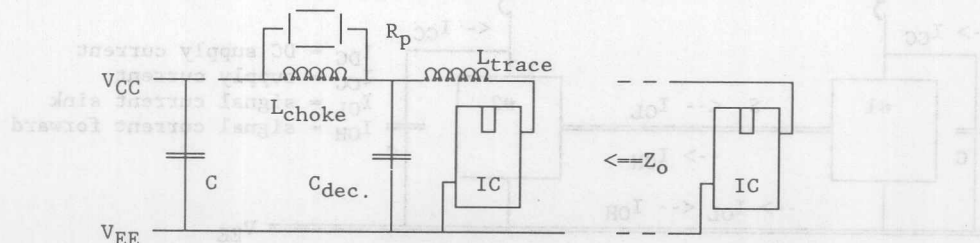


Figure 6. Suggested decoupling circuit with each IC.

Electro magnetic compatibility and printed circuit board (PCB) constraints

ESG89001

The choke may never have an open core, because then it will either act as a RF-transmitter or a ferroreceptor for magnetic fields!!

Example: $1 \text{ MHz} \times 1 \mu\text{H} \rightarrow Z_L = 6.28 \Omega \rightarrow R_s = 3.14 \Omega$
 $Q \leq 2 \quad R_p = 12.56 \Omega$

Above the resonance frequency, the characteristic impedance, Z_0 , of the "transmissionline" (in this case the impedance the IC sees at its supply terminals) will be equal to:

$$Z_0 = \sqrt{(L_{\text{trace}} / C_{\text{decoupling}})}$$

The series inductance of the decoupling capacitor and the inductance of the interconnecting traces have a negligible effect on the RF supply-current distribution, when a choke of e.g. $1 \mu\text{H}$ is used. Still it determines the voltage fluctuations between the supply pins of the IC. With a 25 % signal-to-noise margin dissipation by the power supply, the recommended maximum inductances, L_{trace} , are given in table 4.

FAMILY NOISE-MARGIN	dI / dt	L_{trace}
volt	mA / ns	nH
CMOS(5V) 1.75	2 100	200.
TTL-LS .4	50 10	20.
TTL-F .4	50 2-3	4.
HCT .7	50 2-3	7.
HC(5V) 1.2	50 2-3	12.
ACT 1.7	175 1-2	2.4

Table 4. Allowed (supply) series inductance.

With the decoupling as suggested in fig. 6 the number of transmissionlines between the two ICs has gone down from 3 to 1, see fig. 7.

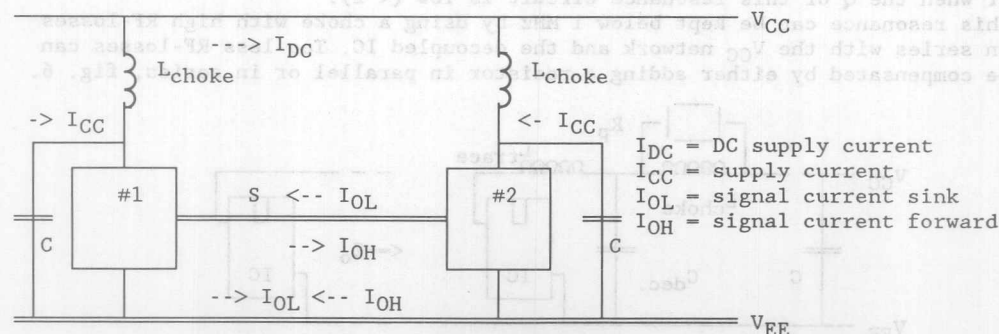


Figure 7. Proper decoupled circuit blocks.

Electro magnetic compatibility and printed circuit board (PCB) constraints

ESG89001

CONCLUSION:

- IV. By using proper decoupling with each IC; $L_{\text{choke}} + C_{\text{dec}}$, only one transmissionline needs to be defined between the circuit blocks.

With high speed logic, $\tau_r < 3$ ns, the total inductance in series with the decoupling capacitor needs to be low, see table 4. A trace, in series with the supply pins, of 50 mm equals an inductance of 50 nH. Together with the load conditions at an output, 50 pF typical, this will give a minimum risetime of 3.2 ns. If faster risetimes are required, shorter leads from the decoupling capacitor (preferred leadless) and shorter leads within the IC-package are necessary. This can be obtained by using e.g. IC-decoupling capacitors, or better, using center (supply) pinned ICs in combination with small leadless ceramic capacitors with a 3E pitch (DIL). A multi-layer board with supply and ground planes can be another option. Further improvements can be reached by applying SO-packages with center pinned supply connections.

CONCLUSION:

- V. When using fast logic; multi-layer panels should be used.

6. Minimize tracelength and limited loopareas.

The maximum tracelength is determined by reflections which will occur at NON-terminated transmissionlines. The loopareas and tracelengths are limited by the EM-radiation which is allowed by mandatory requirements for the product. The latter requirements will directly apply to the PCB if it is used in an unshielded box /cover.

6.1. Allowed tracelengths due to reflections

The first limitation of the tracelength is determined by functional requirements. A transmissionline can be made reflection free by either adding a load resistor at the end of the line, which without series capacitance will cause DC-dissipation, or by adding a resistor in series with the driver. In this case the outputimpedance of the circuit plus the series resistor must be equal to the characteristic impedance of the transmissionline.

When the transmissionline is NOT terminated the allowed trace length is determined by the noise-margin of the logic used, its bandwidth and the propagation delay of the line, which is assumed to be 5 ns/m. The bandwidth determines the dynamic noise margin which by approximation is inverse proportional to the disturbance pulse halfwidth-time. Applying the requirement that the noise, in this case the reflected signal, has to be less then 25 % of the (dynamic) noise-margin the tracelengths in table 5 result.

Electro magnetic compatibility and printed circuit board (PCB) constraints

ESG89001

FAMILY	NOISE-MARGIN volt	dt ns	Maximum tracelength [m]	
			NON-terminated	Series terminated
CMOS	1.75	100	14.3	∞*
TTL-LS	.4	10	0.4	0.5
TTL-F	.4	2-3	0.08	0.15
HCT	.7	2-3	0.14	∞
HC	1.2	2-3	0.24	∞*
AC	1.7	1-2	0.18	∞*

Table 5. The allowed NON- or series-terminated tracelength.

*) If series termination is used in an a-synchronous logic circuit design, attention must be given to the occurrence of meta-stability; especially symmetrical logic input-circuitry cannot decide whether the input signal is high or low and a non-defined output status may /will result.

CONCLUSION:

- VI. A transmission line should, if necessary, be series-terminated at the drivers-side. If the trace lengths are long compared to those given in the table END-termination is inevitable.

6.2. Allowed loop areas due to radiation.

The emission from a PCB (or a complete product) is limited to $100 \mu\text{V/m}$ at 10 meters distance from the object at frequencies above 30 MHz [FCC, IEC, CISPR publications, class B]. This emission is determined by the product of the loop area, A , the loop current, I , and the permeability of the medium within that loop, μ_r (commonly equal to 1). This product is called the magnetic dipole-moment, M .

In case a number of loops are present, operating at the same frequency or clock-rate, the limit of the dipole-moment strength should be divided by \sqrt{n} , in which n = number of loops, hence the signals will add as random noise.

$$M(\text{freq}) = I(\text{freq}) \cdot A \cdot \mu_r$$

The limit value for the magnetic dipole-moment can be calculated from the radiated power [7,8]:

$$E = (7 / r) \cdot \sqrt{P_{\text{rad}}}$$

$$P_{\text{rad}} = 31200 \cdot I^2 \cdot A^2 / \lambda^4 = 31200 \cdot M^2 / \lambda^4$$

where: I = loop current as function of frequency
 A = loop area
 λ = wavelength belonging to the frequency component of the loop current

Electro magnetic compatibility and printed circuit board (PCB) constraints

ESG89001

By substitution the following results:

$$E = (7 / r) \cdot 176 \cdot I \cdot A / \lambda^2.$$

Filling in the requirement, given above, that $E \leq 100 \mu\text{V/m}$ at 10 meters distance from the source the following equation results for the looparea and current as function of frequency:

$$I \cdot A / \lambda^2 \leq 8.1 \cdot 10^{-7} \text{ [A]}, \text{ or}$$

$$M \leq 8.1 \cdot 10^{-7} \cdot \lambda^2 \text{ [A.m}^2 \text{]}$$

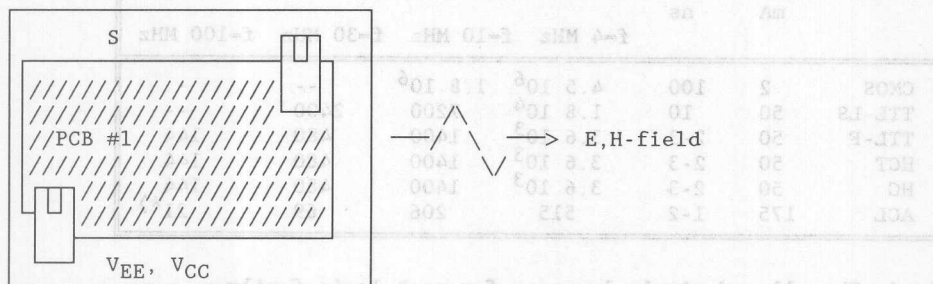


Figure 8. Radiation from a loop on a PCB.

The spectral current amplitude, for logic signals in the frequency domain, decrease above the bandwidth of the logic ($= 1 / \pi \cdot \tau_r$) proportional with frequency square. At this corner frequency, the radiation resistance of the loop still increases proportional with frequency square. Therefore one can calculate the maximum looparea which is determined by the clockrate or repetition rate, the risetime or bandwidth of the logic and the current-amplitude in the time-domain. The current waveshape is derived from the voltage waveshape and the current halfwidth time is by approximation equal to the voltage risetime, fig.9.

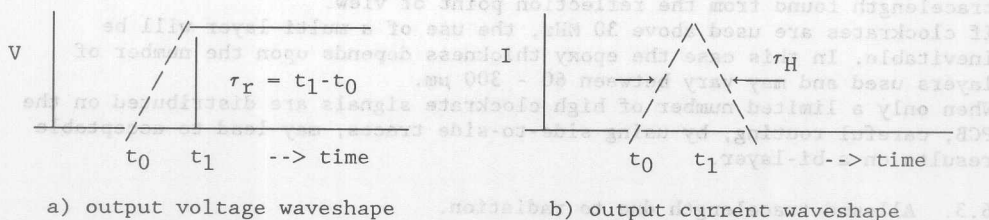


Figure 9. Logic output voltage and current wave shapes in case of capacitive loading.

Electro magnetic compatibility and printed circuit board (PCB) constraints

ESG89001

The current amplitude at the cornerfrequency ($= 1 / \pi \cdot \tau_r$) becomes:

$$I(f) = 2 \cdot I \cdot \tau_r / T$$

where: I = current amplitude in the timedomain,

$T = 1 / \text{clockrate} = \text{period time}$,

τ_r = voltage risetime $\approx \tau_H$ current halfwidth time.

From this equation the maximum looparea at a clockrate for a certain logic family can be calculated. These loopareas are given in table 6.

FAMILY	dI mA	dt ns	Maximum looparea in mm ² at clockrate of			
			f=4 MHz	f=10 MHz	f=30 MHz	f=100 MHz
CMOS	2	100	4.5 10 ⁶	1.8 10 ⁶	--	-
TTL-LS	50	10	1.8 10 ⁴	7200	2400	--
TTL-F	50	2-3	3.6 10 ³	1400	480	144
HCT	50	2-3	3.6 10 ³	1400	480	144
HC	50	2-3	3.6 10 ³	1400	480	144
ACL	175	1-2	515	206	69	21*)

Table 6. The allowed single looparea for each logic family.

*) In this case, when using common DIL-packages, the looparea limit will be exceeded and additional shielding measures, together with proper filtering will be inevitable.

CONCLUSION:

VII. The maximum looparea is determined by the clockrate, the logic family (= output current) and the number, n, of simultaneous switching loops on that PCB.

When a bi-layer is used with a thickness of 1.5 mm, the maximum allowable tracelength, derived from the looparea, will be much less than the tracelength found from the reflection point of view.

If clockrates are used above 30 MHz, the use of a multi-layer will be inevitable. In this case the epoxy thickness depends upon the number of layers used and may vary between 60 - 300 μm .

When only a limited number of high clockrate signals are distributed on the PCB, careful routing, by using side-to-side traces, may lead to acceptable results on a bi-layer.

6.3. Allowed tracelength due to radiation.

The allowed tracelength are even less, when the transmissionline is directly coupled to the system reference and an unshielded outgoing cable leaving the product, then the values found up to now. A simple diagram is given in figure 10. The voltagedrop between the two references with each IC has become the driving source of the antenna formed by reference system and the

Electro magnetic compatibility and printed circuit board (PCB) constraints

ESG89001

outgoing cable. The worst case radiation resistance of the antenna is assumed to be 150Ω and frequency independent [7]. The amplitude of the driving source, U , is now limited to:

$$P_{\text{rad}} = U^2 / 150 \Omega.$$

Applying the radiation requirements as given earlier the voltagedrop has to be:

$$U \leq 1.75 \text{ mV}.$$

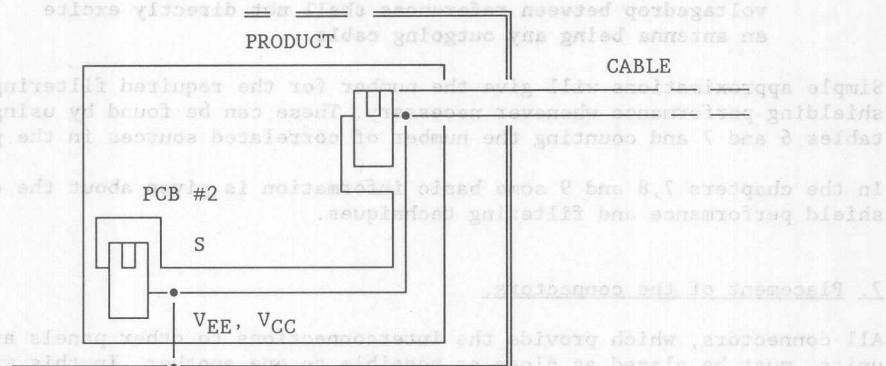


Figure 10. Radiation from a product, containing a PCB, with an outgoing cable.

The voltagedrop is determined by the current amplitude, at the logic bandwidth's frequency, and the effective inductance, see par. 2, of the transmissionline between these points.

$$U(f) = I(f) \cdot Z(f) = I(f) \cdot j \cdot \omega \cdot (L-M) = I(f) \cdot j \cdot \omega \cdot L \cdot (1-k).$$

Taking table 1 and the current amplitude in the frequency domain, the trancelengths in table 7 can be found.

FAMILY	dI mA	dt ns	Allowed trancelength in mm			
			bi-layer / multi-layer			
			f=4 MHz	f=10 MHz	f=30 MHz	f=100 MHz
CMOS	2	100	108/-	44/-	--	--
TTL-LS	50	10	4.3/-	1.75/-	.6/-	--
TTL-F	50	2-3	4.3/55	1.75/40	.6/4.4	-/2.2
HCT	50	2-3	4.3/55	1.75/40	.6/4.4	-/2.2
HC	50	2-3	4.3/55	1.75/40	.6/4.4	-/2.2
ACL	175	1-2	-/15.4	-/3.2	-/2.1	-/0.62

Table 7. Maximum trancelength in case of direct radiation.

Electro magnetic compatibility and printed circuit board (PCB) constraints

ESG89001

This table shows that many practical applications shall not fulfill the radiation requirements.

In most cases, filtering or shielding of the outgoing cable, which leaves the product will be sufficient. Shielding of the entire product, plus the necessary filtering, becomes inevitable when the magnetic loop constraints are exceeded.

CONCLUSION:

VIII. Circuit designs shall be made in such a way that the voltage drop between references shall not directly excite an antenna being any outgoing cable.

Simple approximations will give the number for the required filtering or shielding performance whenever necessary. These can be found by using the tables 6 and 7 and counting the number of correlated sources in the product.

In the chapters 7,8 and 9 some basic information is given about the cable shield performance and filtering techniques.

7. Placement of the connectors.

All connectors, which provide the interconnections to other panels and/or units, must be placed as close as possible to one another. In this way common-mode currents, which are induced in those cables, will NOT flow through the traces of the circuit on the PCB. In addition voltage drop between references on the PCB will not excite the (antenna)-cables.

To avoid such common-mode effects, it may be necessary to make a separation between the reference-strip near to the connectors and the groundplane, groundgrid or reference of the circuitry on the PCB. This groundstrip shall, if applicable, be connected to the metal cover of the product. From this separate groundstrip, only high impedances; inductors, resistors, reed relays and opto-couplers are allowed in-between these two grounds. This will be explained when the filter networks are described, chapter 9.

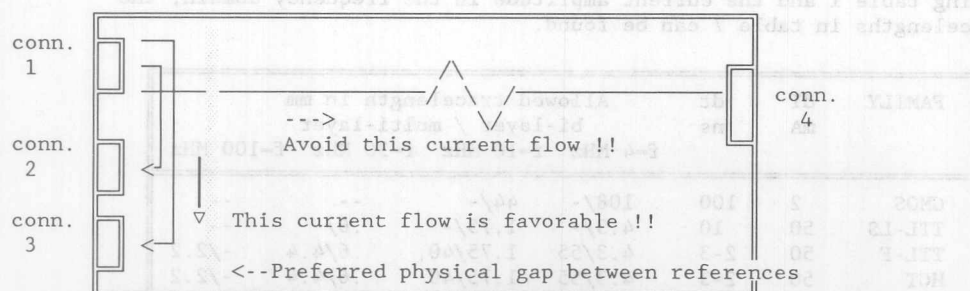


Figure 11. Optimal connector placement on a PCB.

Electro magnetic compatibility and printed circuit board (PCB) constraints

ESG89001

CONCLUSION:

- IX. All connectors need to be placed as close as possible to one another in order to prevent external currents running through the traces or reference of the PCB.

8. Right cable choice with a proper connector.

Cables have, when they are shielded, a transfer impedance, see appendix B. Determined by the amplitude and the frequency content of the signals flowing through these cables a choice shall be made. In case cables, leaving the enclosure of the product, contain data above a 10 kHz clockrate, shielding will be inevitable (product requirement). This shielding shall be connected to ground (metal cover product) on both ends of the cable, this to assure that the shield acts both as an electric and a magnetic shield.

If separate grounds are used, this shall be done to the "connector-ground" instead of the "circuit-ground".

In case the clockrate is above 10 kHz and below 1 MHz and the risetime of the logic is kept as slow as possible, an optical coverage of 80 % or more or a transfer impedance which equals less than 10 nH/m will do. Above 1 MHz clockrates, better shielded cables are always necessary.

In general, coaxial cable excluded; the shield of the cable shall not be used as signal return.

By using passive filters in series with the signal input /outputs to the ground/reference, to reduce the RF-content, the necessity of a high quality shielding and the corresponding connector can be avoided.

A proper shielded cable will have a transfer impedance equal to or less than $|j\omega \cdot 10 \text{ nH/m}|$. Every wire has an inductance of 1 nH/mm, chapter 2.1. In case the shielding of such a cable is wrapped into a pigtail, the inductance of that pigtail will degrade the shielding performance, thus increase the transfer impedance, of the cable.

CONCLUSION:

- X. A good shielded cable deserves a proper connector.

9. Proper use and placement of filters and filterparts.

Signal bandwidth reduction shall be achieved by using RC low-pass filters. In case the voltage drop across the series resistor is unacceptable an inductor with high RF-losses shall be used. The LC low-pass filter will always show resonances and therefore its Q must be kept low.

Electro magnetic compatibility and printed circuit board (PCB) constraints

ESG89001

The filter can be used in two directions namely; to prevent emission from the PCB and to improve the immunity of the board to external sources, e.g. RF-transmitters, ESD, etc.

The lay-out of the interconnection of the shield of the cable and a low-pass RCR-filter is given in figure 12. The lay-out of the filter shall be such that the requirements for the maximum loop areas, table 6, and the requirements for the maximum tracelength, table 7, are not violated.

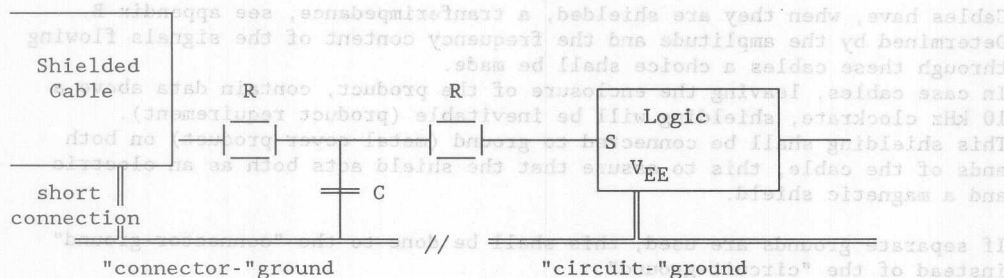


Figure 12. Lay-out of the interconnection and filtering of a shielded cable to a PCB.

CONCLUSIONS:

XI. Currents, which do not belong to the circuit signals, should be by-passed using another path.

XII. The bandwidth of signals should be limited to the least functional bandwidth. Use the slowest logic family suitable for the function.

A proper shielded cable will have a transimpedance equal to or less than 10 mΩ. Every wire has an inductance of 1 nH/mm, chapter 2.1. In case the shielding of such a cable is wrapped into a pigtail, the inductance of that pigtail will degrade the shielding performance, thus increase the transimpedance of the cable.

CONCLUSION:

X. A good shielded cable deserves a proper connector.

9. Proper use and placement of filters and filter parts

Signal bandwidth reduction shall be achieved by using RC low-pass filters. In case the voltage drop across the series resistor is unacceptable an inductor with high RF-losses shall be used. The RC low-pass filter will always show resonances and therefore its Q must be kept low.

Electro magnetic compatibility and printed circuit board (PCB) constraints

ESG89001

10. PCB demo-board, routing and decoupling effects.

An EURO-card PCB (100 x 160 mm²) has been chosen to demonstrate the effects of signal lines and their signal returns with respect to magnetic radiation.

The board contains a relaxation oscillator, created by 3 inverters (NANDS) and an RC-network (1 k Ω , 560 pF), which will produce a squarewave voltage signal. The frequency will be determined by the used logic and its threshold voltages. This oscillator is placed in one corner of the board together with some switches to change signal-return path and supply decoupling. In the opposite corner of the PCB another quad NAND has been placed as a capacitive load. These NANDS are all cascaded and will change status with some skew. The last NAND is terminated by a resistor. The supply decoupling of this IC can be altered as well.

The diagram of the circuit with the switches is given in figure 13 and the physical lay-out of the PCB and component placement are given in figure 14.

The layout has been chosen such that the supply traces are as close as possible to one-another, which is commonly arranged by a proper CAD-tool. In parallel to the signal trace a signal return trace has been placed, according to chapter 4. At the supply pins of the ICs decoupling capacitors are added to each IC. By means of jumpers or switches a series inductor may be short-circuited or added to the circuit.

In total 4 relevant situations can be evaluated which are given in the table 8.

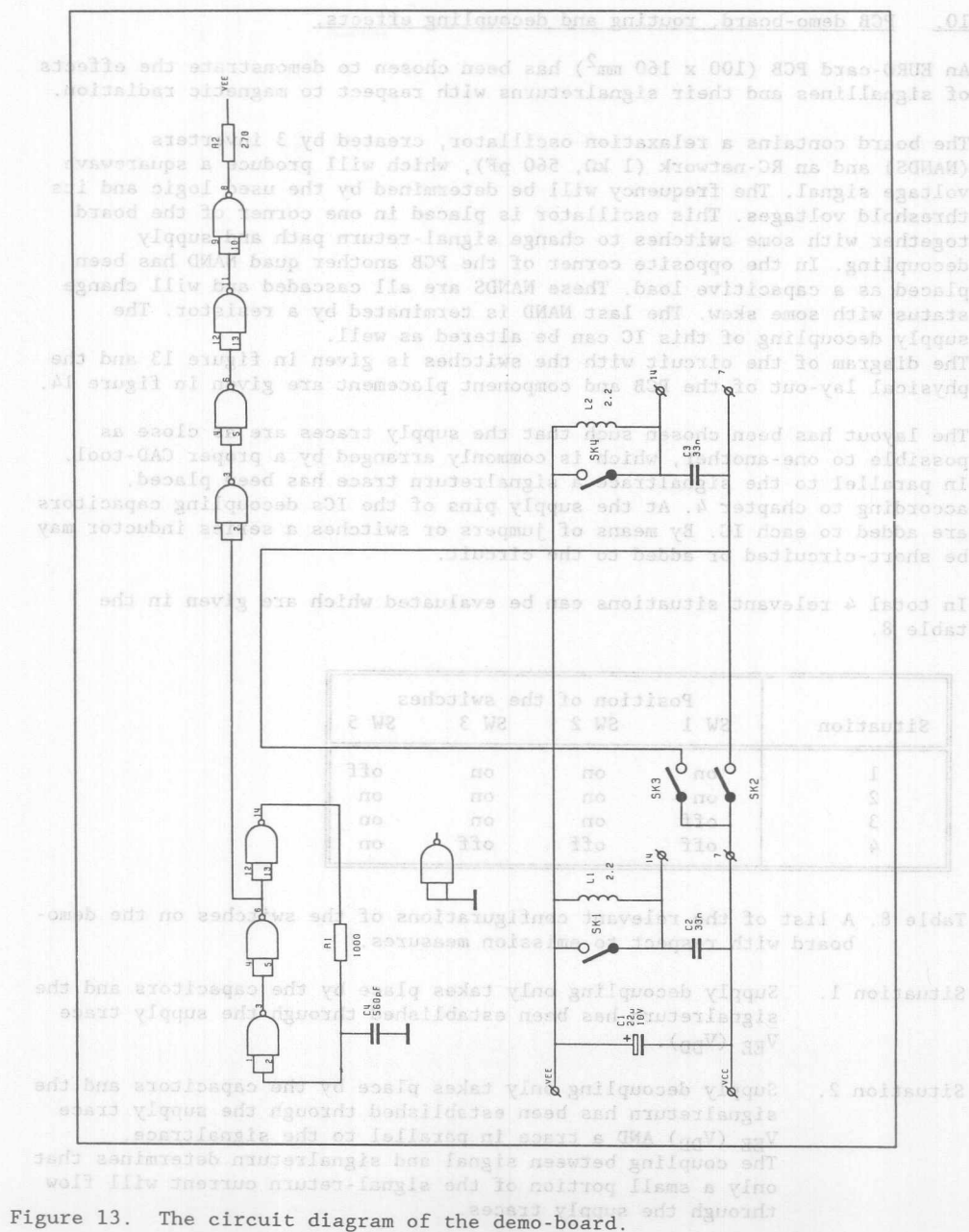
Situation	Position of the switches			
	SW 1	SW 2	SW 3	SW 5
1	on	on	on	off
2	on	on	on	on
3	off	on	on	on
4	off	off	off	on

Table 8. A list of the relevant configurations of the switches on the demo-board with respect to emission measures.

- Situation 1. Supply decoupling only takes place by the capacitors and the signal return has been established through the supply trace V_{EE} (V_{DD}).
- Situation 2. Supply decoupling only takes place by the capacitors and the signal return has been established through the supply trace V_{EE} (V_{DD}) AND a trace in parallel to the signal trace. The coupling between signal and signal return determines that only a small portion of the signal-return current will flow through the supply traces.
- Situation 3. The supply trace, V_{EE} (V_{DD}), has been taken out and the I_{CC} and signal-return-current have to flow through the trace next to the signal line. The high frequency components of the

Electro magnetic compatibility and printed circuit board (PCB) constraints

ESG89001



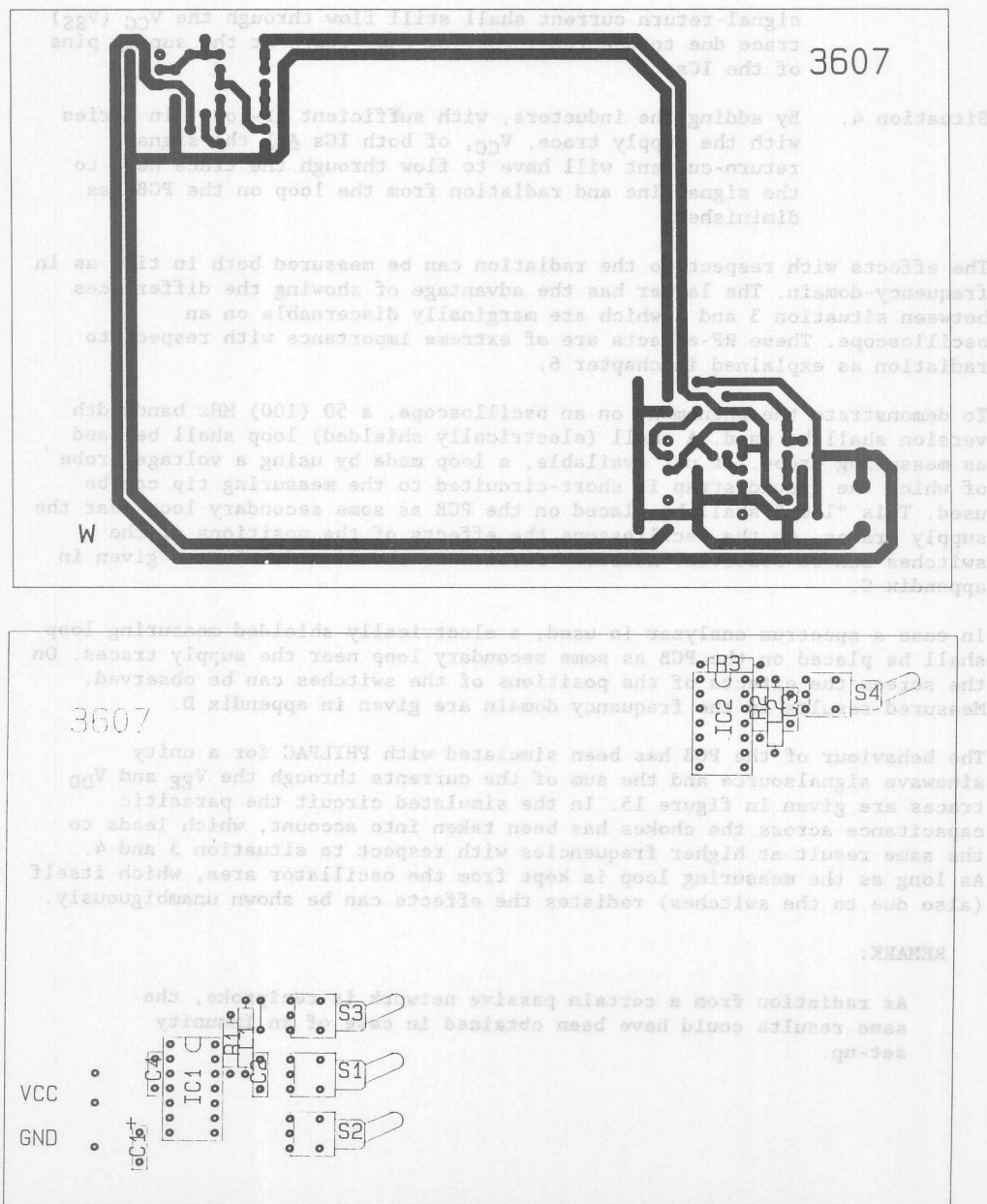


Figure 14. The lay-out and components placement of the demo-board.

Electro magnetic compatibility and printed circuit board (PCB) constraints

ESG89001

signal-return current shall still flow through the V_{CC} (V_{SS}) trace due to the (de-)coupling capacitors at the supply pins of the ICs.

Situation 4. By adding the inductors, with sufficient RF-loss, in series with the supply trace, V_{CC} , of both ICs ALL the signal-return-current will have to flow through the trace next to the signalline and radiation from the loop on the PCB has diminished.

The effects with respect to the radiation can be measured both in time as in frequency-domain. The latter has the advantage of showing the differences between situation 3 and 4 which are marginally discernable on an oscilloscope. These RF-effects are of extreme importance with respect to radiation as explained in chapter 6.

To demonstrate the phenomena on an oscilloscope, a 50 (100) MHz bandwidth version shall be used. A small (electrically shielded) loop shall be used as measuring probe. If not available, a loop made by using a voltage probe of which the ground strap is short-circuited to the measuring tip can be used. This "loop" shall be placed on the PCB as some secondary loop near the supply traces. On the oscilloscope the effects of the positions of the switches can be observed. Measured results in the time domain are given in appendix C.

In case a spectrum analyzer is used, a electrically shielded measuring loop shall be placed on the PCB as some secondary loop near the supply traces. On the screen the effects of the positions of the switches can be observed. Measured results in the frequency domain are given in appendix D.

The behaviour of the PCB has been simulated with PHILPAC for a unity sinewave signalsource and the sum of the currents through the V_{EE} and V_{DD} traces are given in figure 15. In the simulated circuit the parasitic capacitance across the chokes has been taken into account, which leads to the same result at higher frequencies with respect to situation 3 and 4. As long as the measuring loop is kept from the oscillator area, which itself (also due to the switches) radiates the effects can be shown unambiguously.

REMARK:

As radiation from a certain passive network is reciproke, the same results could have been obtained in case of an immunity set-up.



Figure 14. The lay-out and component placement of the demo-board.

Electro magnetic compatibility and printed circuit board (PCB) constraints

ESG89001

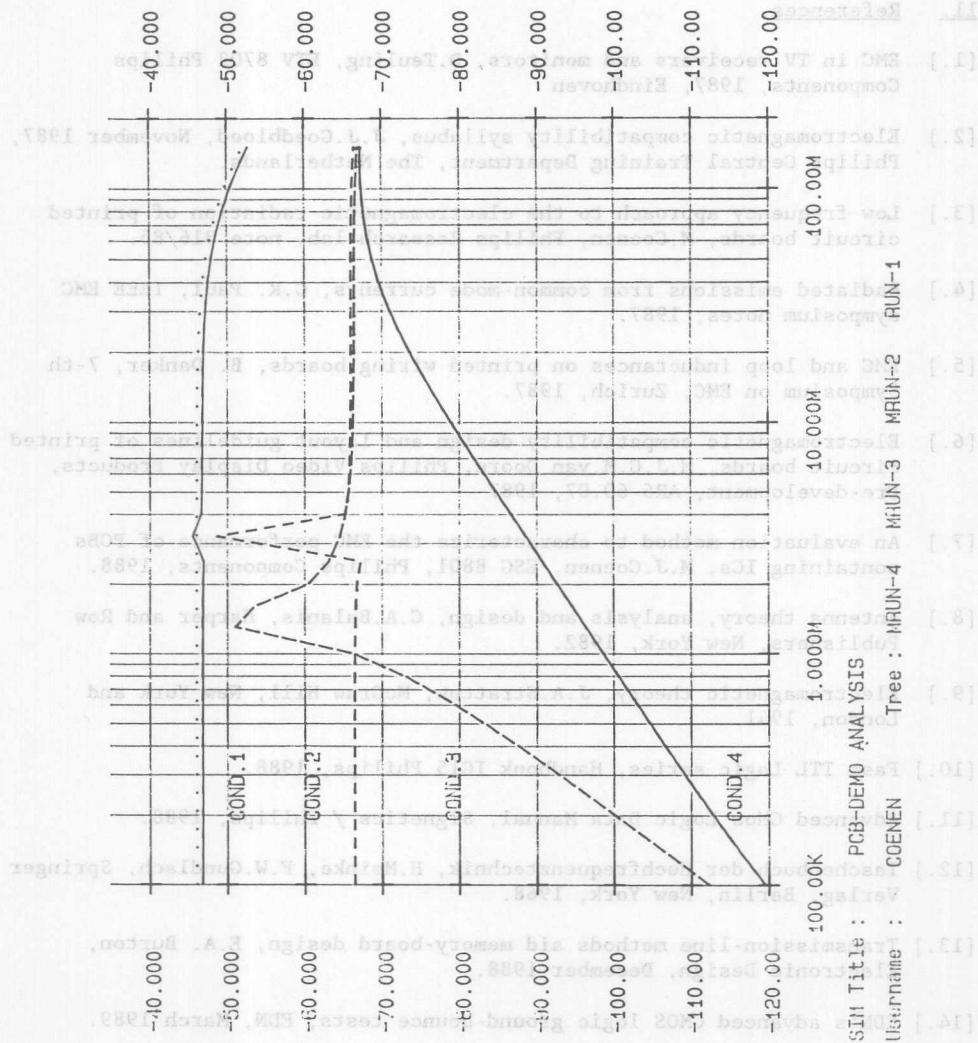


Figure 15. PHILPAC AC-analysis of the EM-radiation behaviour of the demo-board in the 4 conditions.

Electro magnetic compatibility and printed circuit board (PCB) constraints

ESG89001

11. References

- [1.] EMC in TV receivers and monitors, D.Teuling, ETV 8702 Philips Components, 1987, Eindhoven
- [2.] Electromagnetic compatibility syllabus, J.J.Goedbloed, November 1987, Philips Central Training Department, The Netherlands.
- [3.] Low frequency approach to the electromagnetic radiation of printed circuit boards, M.Coenen, Philips Research lab. note 316/85.
- [4.] Radiated emissions from common-mode currents, C.R. Paul, IEEE EMC symposium notes, 1987.
- [5.] EMC and loop inductances on printed wiring boards, B. Danker, 7-th Symposium on EMC, Zurich, 1987.
- [6.] Electromagnetic compatibility design and layout guidelines of printed circuit boards, M.J.C.M.van Doorn, Philips Video Display Products, Pre-development, AR6-60.07, 1987
- [7.] An evaluation method to characterize the EMC performance of PCBs containing ICs, M.J.Coenen, ESG 8801, Philips Components, 1988.
- [8.] Antenna theory, analysis and design, C.A.Balanis, Harper and Row Publishers, New York, 1982.
- [9.] Electromagnetic theory, J.A.Stratton, McGraw Hill, New York and London, 1941.
- [10.] Fast TTL Logic series, Handbook IC15 Philips, 1988.
- [11.] Advanced CMOS Logic Data Manual, Signetics / Philips, 1988.
- [12.] Taschenbuch der Hochfrequenztechnik, H.Meinke, F.W.Gundlach, Springer Verlag, Berlin, New York, 1968.
- [13.] Transmission-line methods aid memory-board design, E.A. Burton, Electronic Design, December 1988.
- [14.] EDN's advanced CMOS logic ground-bounce tests, EDN, March 1989.

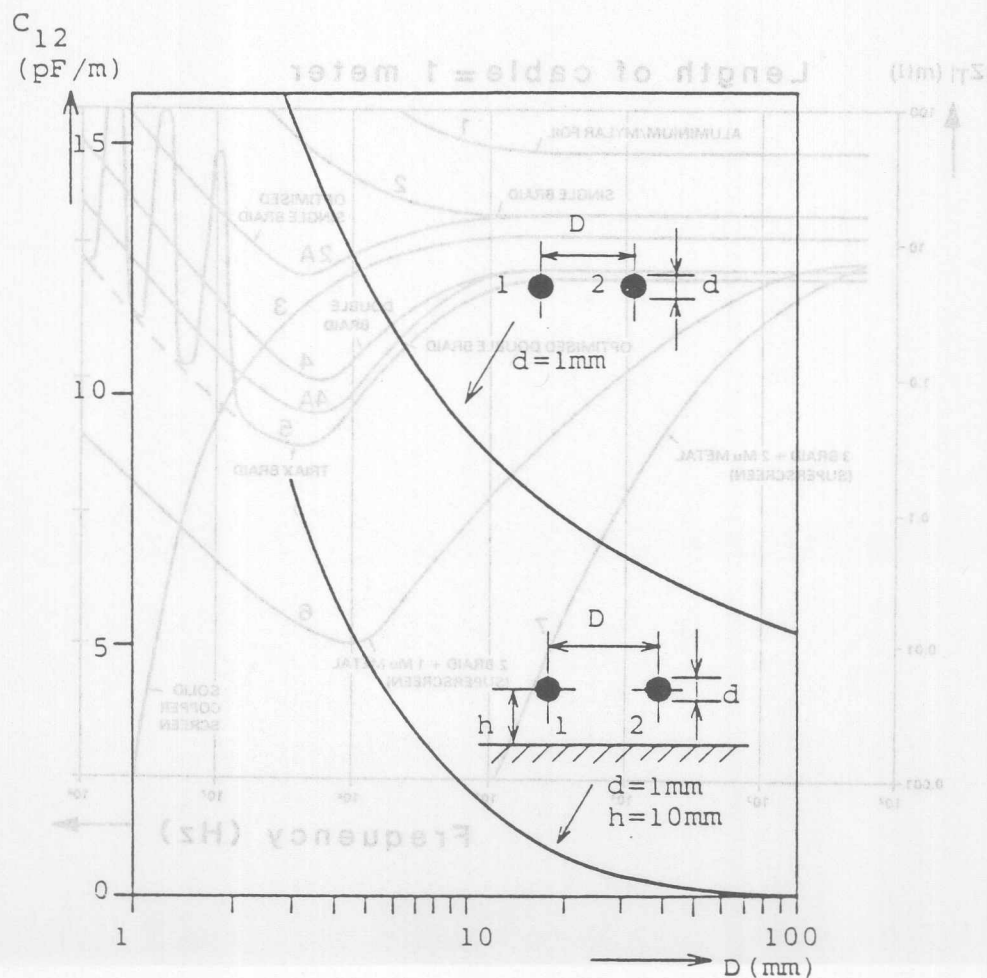
Electro magnetic compatibility and printed circuit board (PCB) constraints

ESG89001

Appendix A Capacitive coupling between traces.

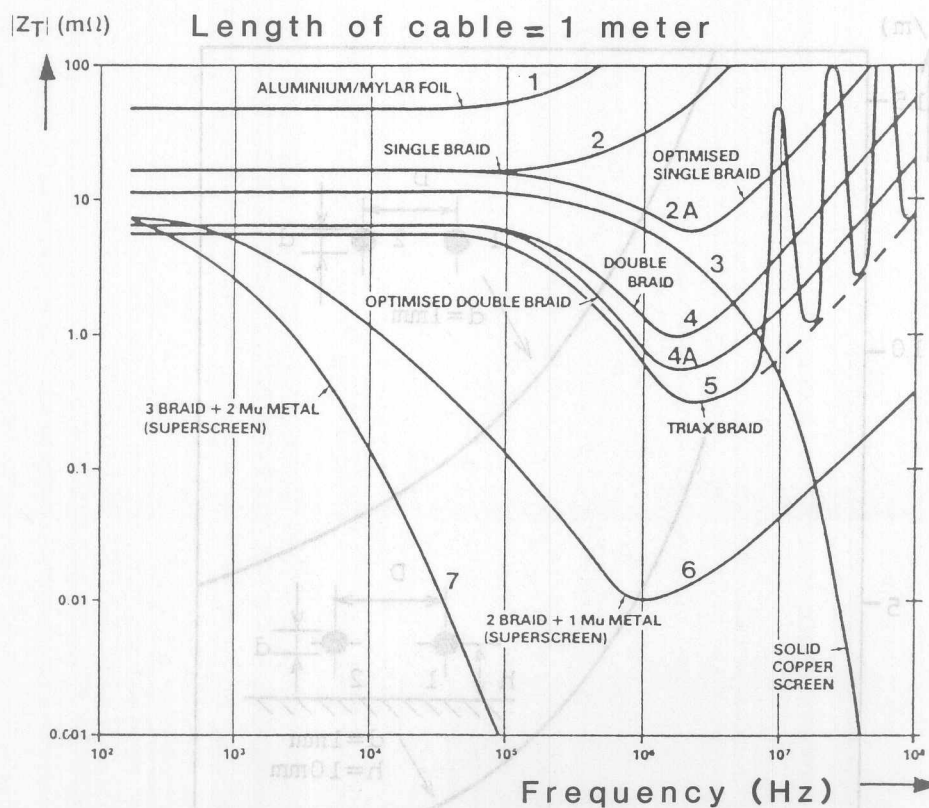
In this appendix the graphical presentation is given of the capacitive coupling between two traces in free space and for two traces above a reference plane [12, form. 24.25].

It shows the necessity of a reference plane at a height, h , closer to the traces than the distance, D , to reduce the capacitive coupling between the traces.



Appendix B The transferimpedance of various cable screens.

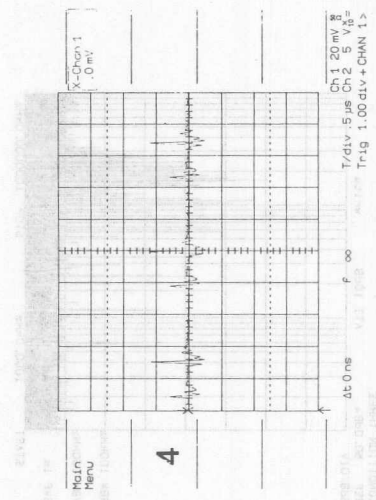
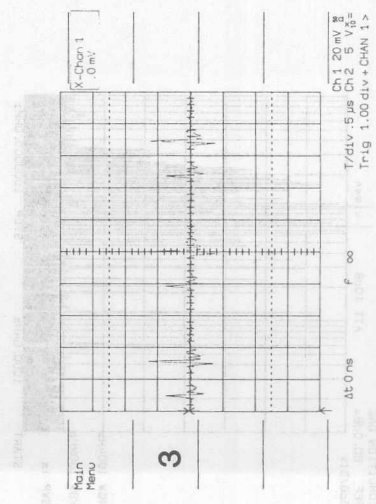
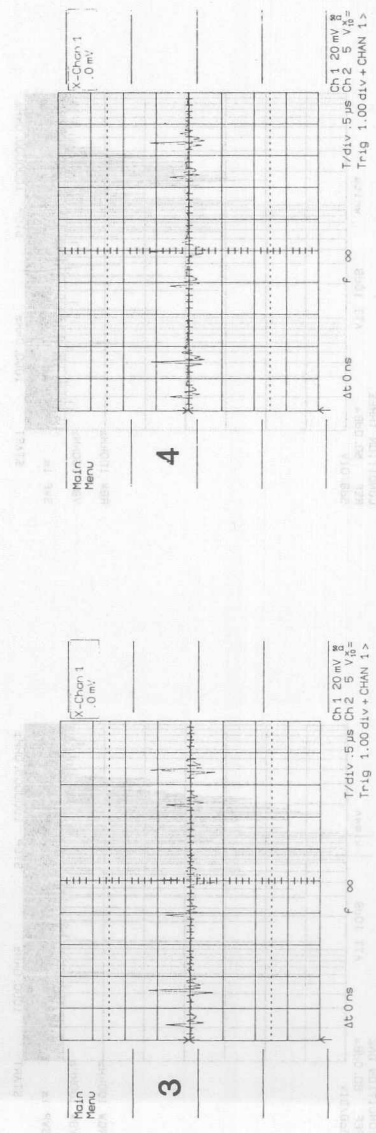
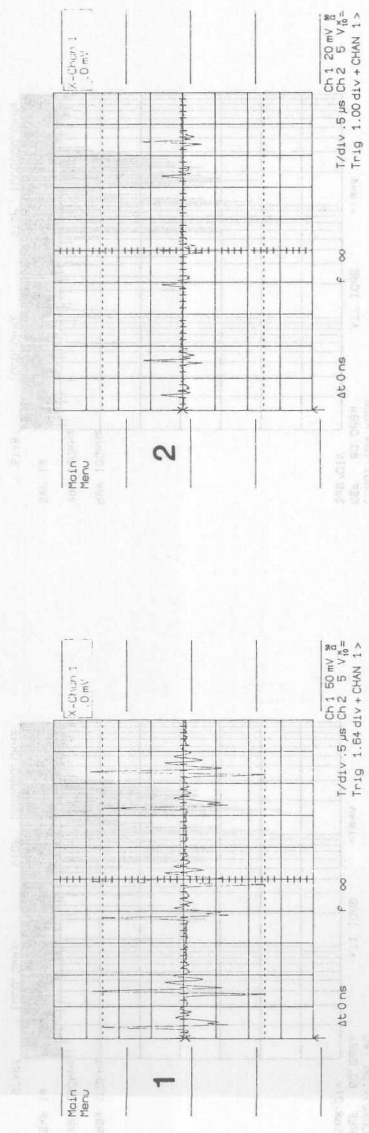
The transferimpedance, Z_t , is the relation between the current through the screen due to an external source and the induced voltage across the nominal load impedances of that cable. Further information about the measuring method to obtain information of the screening efficiency or the transferimpedance can be found in IEC publication 96.



Electro magnetic compatibility and printed circuit board (PCB) constraints

ESG89001

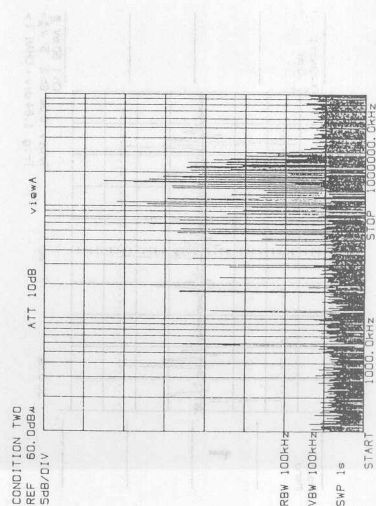
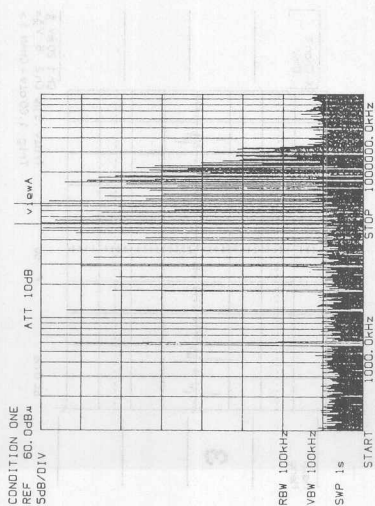
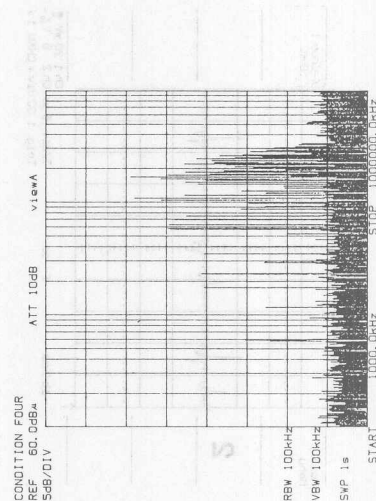
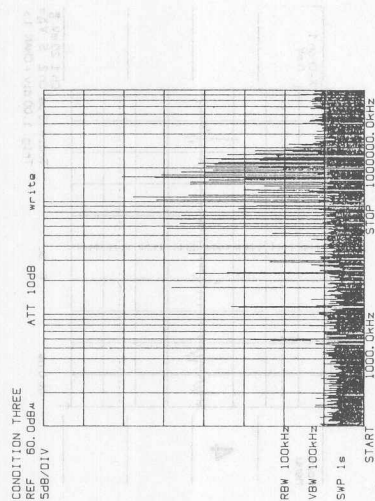
Appendix C Measured results in the time domain, 150 MHz bandwidth, from the demo-board, containing a 74HCT00, in the 4 conditions described in chapter 10.



Electro magnetic compatibility and printed circuit board (PCB) constraints

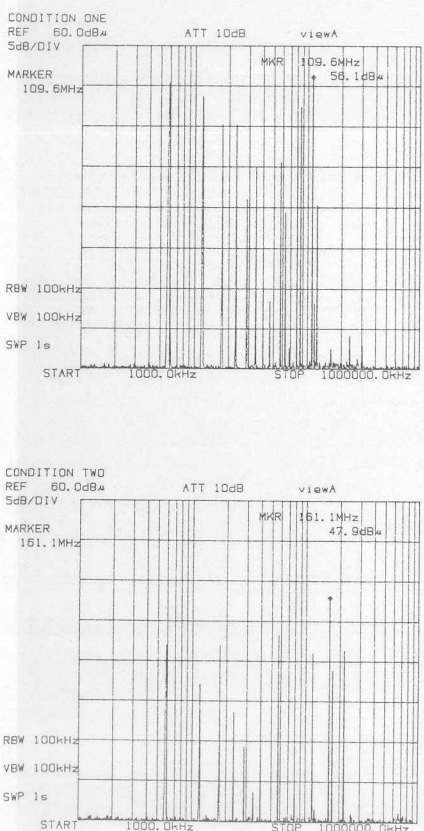
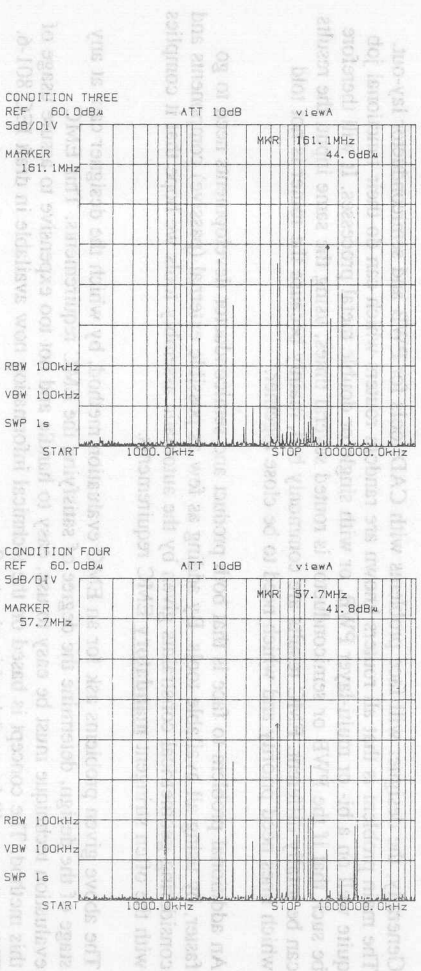
ESG89001

Appendix D1 Measured results in the frequency domain, PEAK detection, from the demo-board, containing a 74HCT00, in the 4 conditions described in chapter 10.



Electro magnetic compatibility and printed circuit board (PCB) constraints

Appendix D2 Measured results in the frequency domain, AVERAGE detection, from the demo-board, containing a 74HCT00, in the 4 conditions described in chapter 10.



Workbench EMC evaluation method

EIE/AN91001

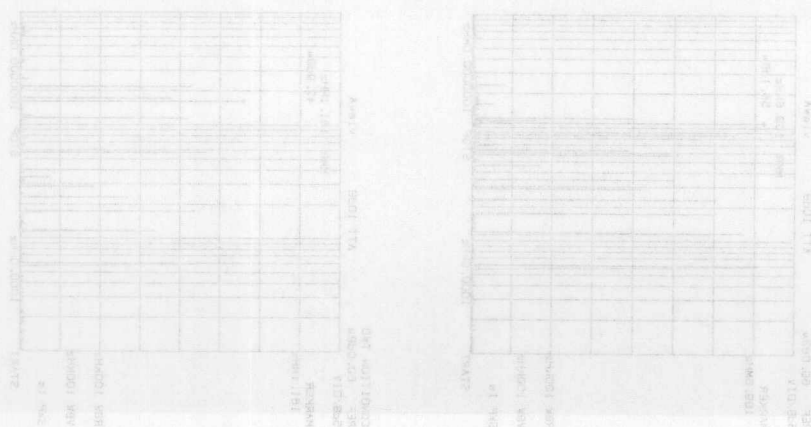
1 INTRODUCTION

In many mass manufactured electrical and electronic products, printed wiring boards (PWBs) are used in a non-shielded application because it would be too expensive to do otherwise, e.g. hand-held cassette players, telephones, televisions, etc. As such, the product related EMC requirements directly apply to the (main) PWB containing semiconductors. To meet economic constraints, EMC solutions need to be taken, as far as possible, within the semiconductors, such that the needs for external measures diminish. In order to select between the various applications suggested by different vendors, an EMC qualification is required too.

Generally, the designer will face problems with CAD tools for PWB and semiconductor lay-out. The main problem is that all routers known are random routers which can do their functional job quite well on a bi- or multi-layer PWB, or with single or double metal processes. It will therefore be such that if the PWB or semiconductor is routed several times, using the same input, the results can be totally different. EMC results are commonly bad, mainly because the router is not told which lines need priority and which need to be close together.

An additional problem to face is that both product and semiconductor developments need to go faster, by using all available tools. By adding as few as possible external (passive) components and considering geometrical constraints given by the automatic assembly tools we hope that it complies with the often stringent mandatory EMC requirements.

The above given problems ask for an EMC evaluation method, by which the designer can, at any stage of the design, determine the degree of satisfying the EMC requirements. This EMC evaluation technique must be easy to use, easy to handle and not too expensive to assure usage of this method. The concept is based on the technical information now available in draft IEC 801-6. This concept will be explained in chapter 2.



Workbench EMC evaluation method

EIE/AN91001

2 TEST METHOD

2.1 EMC regulations and standards

With respect to EMC-standards we can concentrate ourselves best to the generic emission and immunity requirements given in European Standards EN 50081-1 and EN 50082-1 which become legally enforceable from 1992 onwards, especially to the European Market. In these standards reference is made to either IEC documents or CENELEC standards in which certain disturbance or emission phenomena are depicted. For most product groups, the IEC CISPR 11/22 documents or CENELEC European Norm 55011/22, class B apply for RF-emission, whereas for immunity phenomena IEC 801-2 through -6 or CENELEC 55024-x apply.

In our particular case, considering IC-development and IC-application, the most important document is draft IEC 801-6 as it can be applied both to immunity and emission. The validity is given by the fact that the EM-radiation properties of cables and wires connected to the Device or Equipment Under Test (DUT or EUT) are substituted by simple passive networks. As passive networks have reciprocity both emission and immunity can be evaluated in the same set-up.

2.2 Basic Concept

With the method, according to draft IEC 801-6, immunity to **conducted** RF-disturbances is tested. The set-up simulates the EM radiation effects by coupling the induced disturbing signals through Coupling/Decoupling Networks (CDNs) via the cables and wires to the PWB under test in a defined way, Fig.1. This method is applicable in the frequency range 150 kHz up to 230 (1000) MHz. The set-up with CDNs simulates "passive" cables which are, from the radiation point of view, at resonant length. The dimension of the PWB(s) between these cables and wires is assumed to be short compared to wavelengths involved.

According to the existing radiation measurement procedures, either the cable contributions are eliminated by adding ferrites on them to serve reproducibility (EN 55020) or the cables shall be adjusted in length and geometry with each frequency to obtain maximum radiation (EN 55011/22). With the conducted "interaction" method, which simulates radiated RF-field phenomena, both problems are solved.

Note 1: This method also reproduces the electric and magnetic near-fields to the PWB, associated with the source of disturbance (E and H in figure 2).

Note 2: The method of publication 801-6 does not provide the injection (or measurement) of the current from an ideal current or voltage source. It rather provides the coupling of the disturbance signal from a real source, having a "radiation" resistance of 150 Ω , i.e. short-circuit current as well as open voltage are limited.

CDNs are mainly defined by the common-mode impedance represented at the EUT-port side, which needs to be about 150 Ω to reference ($\pm 20 \Omega$, freq. ≤ 30 Mhz, $+60/-45 \Omega$, freq. > 30 Mhz). Every CDN fulfilling the impedance requirements given in draft IEC 801-6 can be used. Typical application examples are given in the figures 3a) for shielded and 3b) for unshielded cables. In figure 3c) a simplified drawing for mechanical construction is given. By using new NiZn ferrite ($\mu_r \geq 1000$), the entire frequency range can be covered by using one toroid on which we have 17 windings, $L_{(150 \text{ kHz})} \geq 280 \mu\text{H}$, and one bead-on-cable only.

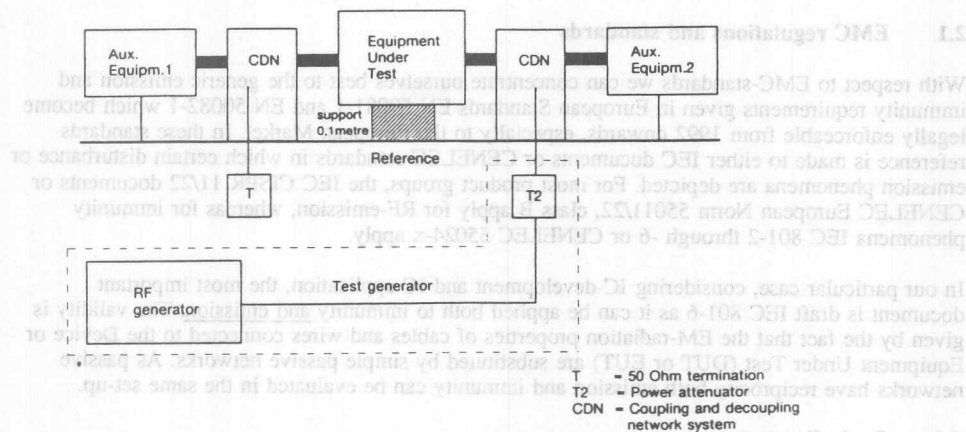


Fig.1. Schematic set-up for immunity test to RF conducted disturbances.

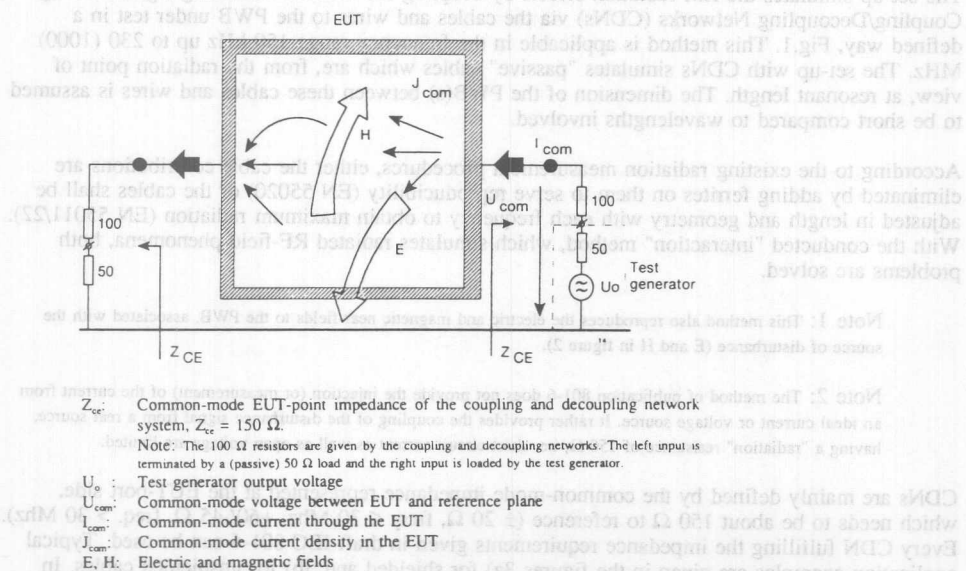


Fig.2. Equivalent circuit of Fig.1 to explain the electromagnetic near-fields approximated by common-mode currents and voltages induced by a RF-source according to the immunity method to conducted disturbances.

Workbench EMC evaluation method

EIE/AN91001

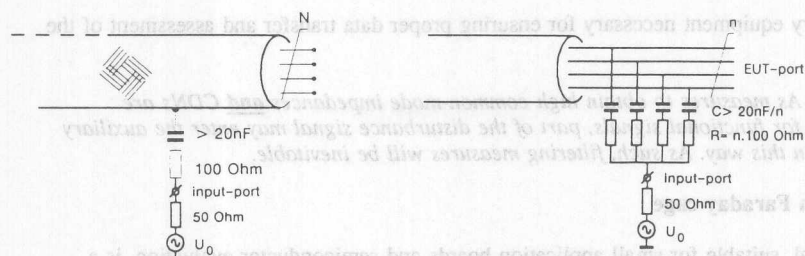


Fig.3a. Coupling to shielded cables Fig.3b. Coupling to un-shielded (multi-wire) cables.

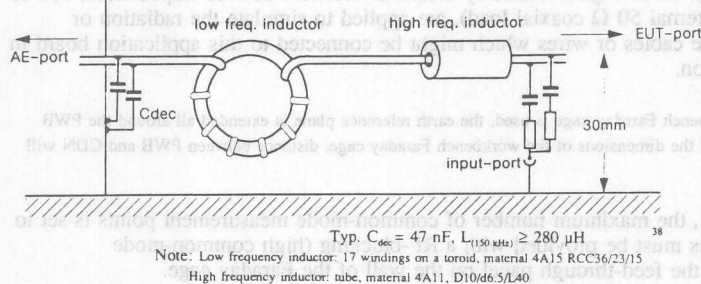


Fig.3c. Mechanical drawing for the Coupling/Decoupling Network (CDN).

2.3 Test set-up for individual PWB units

According to draft IEC 801-6, the PWB is placed on an insulating support, 0,1 metre above an earth reference plane. If the PWB will be used in a cabinet where a metal part is more near to the PWB than 0,1 metre, e.g. an adjacent PWB or a metal floor plate, then that shorter distance must be used. The earth reference plane shall exceed the projected geometry of the PWB and the used CDNs on all sides by at least 0,2 m.

On all cables measures to obtain high common-mode impedances at the frequencies of interest or CDNs must be inserted. The number of CDNs should be limited (between 3 and 5) by restricting oneself to the representative functions and main (disturbing) current distributions occurring to the application in practice. The CDNs need to be placed directly on the earth reference plane, making proper contact to it, at a maximum distance of 0,3 metre from the PWB. The cables between the CDNs and the PWB shall be as short as possible. Their height over the earth reference plane must be kept between 30 and 50 mm (as long as possible).

All auxiliary equipment, AE, required for the defined operation of the PWB, according to the specifications of the product, must be connected through high common-mode impedances or through CDNs to the PWB, e.g. communication, modem, printer, sensor, etc. This shall also be

Workbench EMC evaluation method

Workbench EMC evaluation method EIE/AN91001

done for all auxiliary equipment necessary for ensuring proper data transfer and assessment of the functions.

*Warning 1: As measures to obtain high common-mode impedances **and** CDNs are transparent for functional signals, part of the disturbance signal may enter the auxiliary equipment in this way. As such, filtering measures will be inevitable.*

2.4 Workbench Faraday cage

The present proposal, suitable for small application boards and semiconductor evaluation, is a table-top size Faraday cage where all the connections to DC-supply and other auxiliary equipment are made through filters mounted on the wall of the Faraday cage. The wires and cables from these filters to the PWB need to be wrapped on a ferrite toroid, Philips NiZn material 4A15, to create a high common-mode impedance towards the walls of the Faraday cage (= earth reference plane), similar to the construction given in Fig.3c. Then, by means of resistors defined impedances, 150 Ω (100 Ω in series with an external 50 Ω coaxial load), are applied to simulate the radiation or reception performance of the cables or wires which might be connected to this application board in practical (product) application.

Note 3: When a workbench Faraday cage is used, the earth reference plane is extended all around the PWB under test. As a result of the dimensions of the workbench Faraday cage, distance between PWB and CDN will remain $\leq 0,3$ metre.

To keep this method simple, the maximum number of common-mode measurement points is set to 3. All other wires and cables must be provided with a RF-blocking (high common-mode impedance) device towards the feed-through panel on the wall of the Faraday cage.

The common-mode measurement points on the PWB selected for evaluation are:

- at the DC-power supply connector,
- at the input port connector and
- at the output port connector.

Dependent on the implementation of the application board, containing one or several ICs, the emission or immunity performance can be measured, Fig.6 and 7. In those set-ups the coaxial 50 Ω loads outside the Faraday cage shall be exchanged in turn with either the selective voltmeter (spectrum analyzer) or the disturbance source.

For worst-case testing the three connector positions, which are selected as common-mode points, are distributed all alongside the PWB, Fig.4a. This arrangement is chosen because one can not predict the geometrical lay-out the customer is going to use in his product.

Note 4: Only if an application is required (to fulfil the requirements) such that all connectors are placed on one side only, Fig.4b, it must be tested as such. As a result, this condition then, shall be clearly stated in the application report.

The determination of the common-mode points of the selected ports is based on the cables or wire-geometries likely to occur in product application. Furthermore, dependent on the product application, the appropriate CDN shall be selected.

Workbench EMC evaluation method

EIE/AN91001

- When shielded cables are used, the shielding is referred to as common-mode point of that port (towards the wall of the Faraday cage), no matter how many wires are within that shielding.
- When unshielded cables are used, the common-mode impedance shall be established by placing a number of resistors to each individual wire such that the total common-mode resistance equals $100\ \Omega$ (+ $50\ \Omega$ external) towards the reference, being the wall of the Faraday cage. In series with those resistors capacitors ($C_{\text{total}} \geq 20\ \text{nF}$) must be applied such that the impedance requirements are still met.

In product applications where an IC is fed by a signal source properly defined, i.e. signal and ground return are adjacent tracks, separation $\leq 1\ \text{mm}$, and the tracks are short, length $\leq 0,1\ \text{metre}$, a coaxial CDN-type can be applied. The same applies for the output configuration. Commonly, supply traces are not kept adjacent properly from IC to supply and therefore an unbalanced two-wire CDN should be used. If an IC is properly decoupled, supply and ground will be RF-short-circuited and a coaxial CDN can be applied.

Warning 2: With analog circuits, outputs often cannot handle high capacitive loading which is represented by CDNs and its cables. As only the demodulated component at 1 kHz is of interest, an RC-filter need to be used in-between output and CDN, see Fig.5.

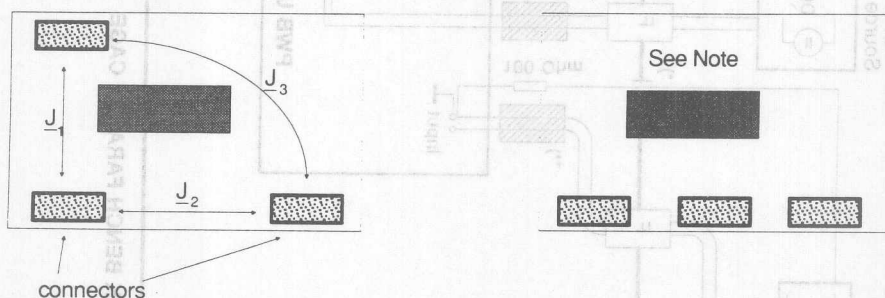


Fig.4. Typical arrangements of inputs, outputs and supply alongside the IC.

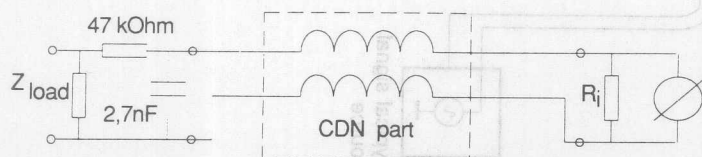
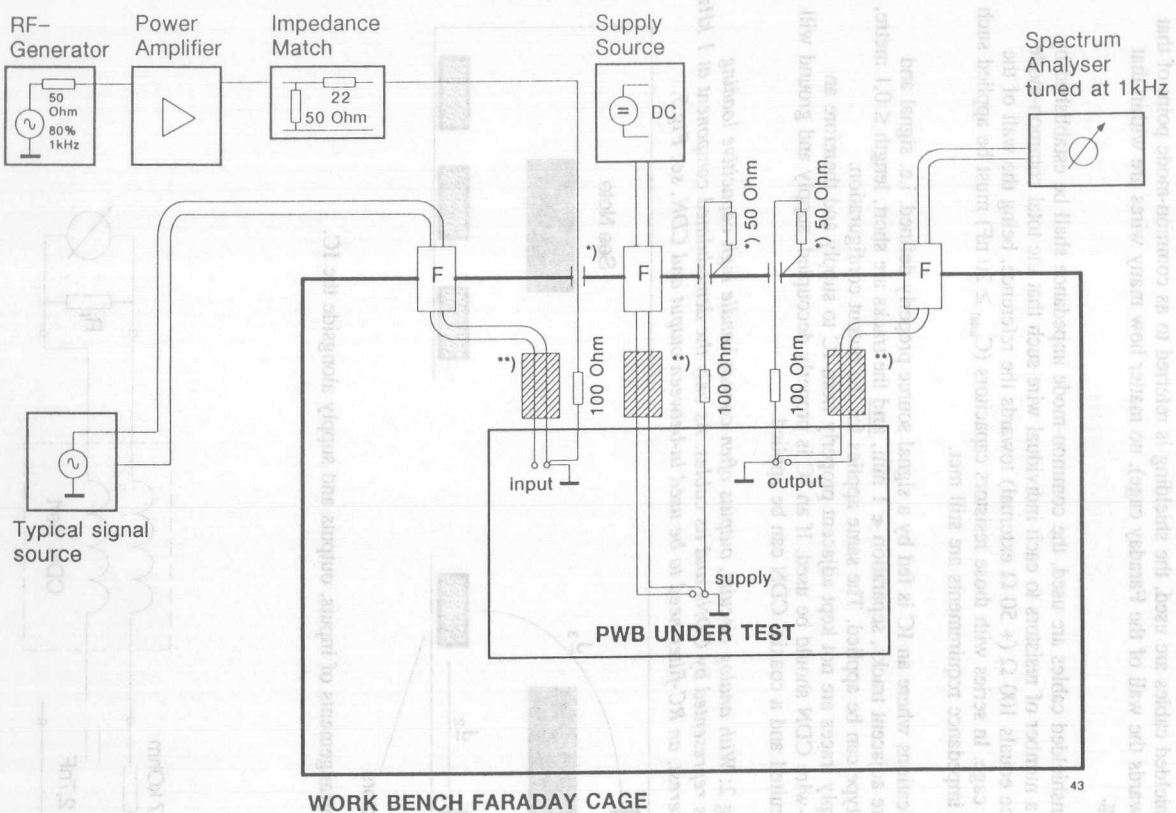


Fig.5. Filter/impedance matching network necessary to lower capacitive loading at IC outputs.

Workbench EMC evaluation method

bottom noiseless 3M EIE/AN91001



- F = Filter
 *) shall be interchanged at all ports
 **) RF-impedance ($\gg 150 \text{ Ohm}$)

Fig. 6. Simplified set-up for immunity testing of the PWB using the workbench Faraday cage.

Workbench EMC evaluation method

EIE/AN91001

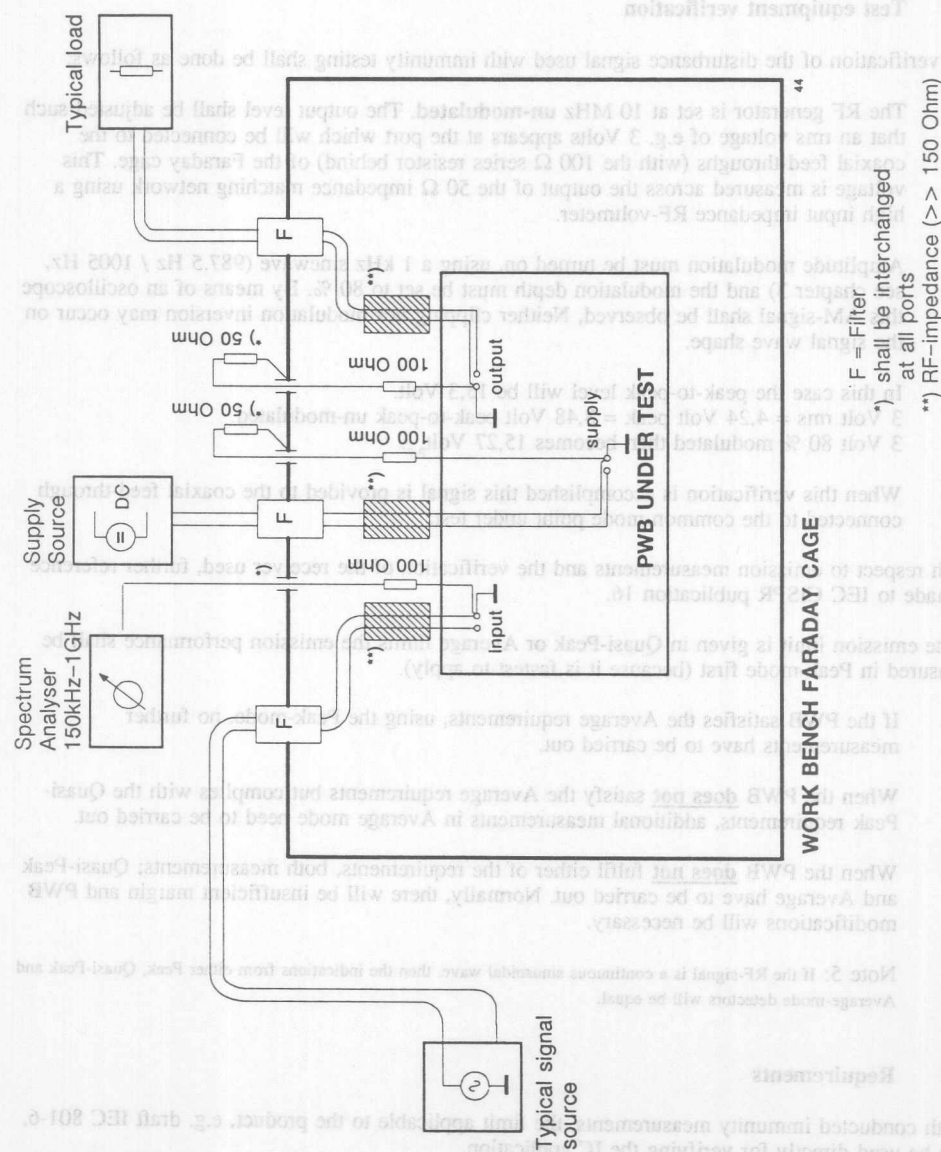


Fig.7. Simplified set-up for emission testing of the PWB using the workbench Faraday cage.

Workbench EMC evaluation method

EIE/AN91001

2.5 Test equipment verification

The verification of the disturbance signal used with immunity testing shall be done as follows:

1. The RF generator is set at 10 MHz **un-modulated**. The output level shall be adjusted such that an rms voltage of e.g. 3 Volts appears at the port which will be connected to the coaxial feed-throughs (with the 100 Ω series resistor behind) of the Faraday cage. This voltage is measured across the output of the 50 Ω impedance matching network using a high input impedance RF-voltmeter.
2. Amplitude modulation must be turned on, using a 1 kHz sinewave (987.5 Hz / 1005 Hz, see chapter 3) and the modulation depth must be set to 80 %. By means of an oscilloscope this AM-signal shall be observed, Neither clipping nor modulation inversion may occur on the signal wave shape.

In this case the peak-to-peak level will be 15,3 Volt.
3 Volt rms = 4,24 Volt peak = 8,48 Volt peak-to-peak un-modulated.
3 Volt 80 % modulated then becomes 15,27 Volt_{p-p}.
3. When this verification is accomplished this signal is provided to the coaxial feed-through connected to the common-mode point under test.

With respect to emission measurements and the verification of the receiver used, further reference is made to IEC CISPR publication 16.

If the emission limit is given in Quasi-Peak or Average limits the emission performance shall be measured in Peak-mode first (because it is fastest to apply).

1. If the PWB satisfies the Average requirements, using the Peak-mode, no further measurements have to be carried out.
2. When the PWB **does not** satisfy the Average requirements but complies with the Quasi-Peak requirements, additional measurements in Average mode need to be carried out.
3. When the PWB **does not** fulfil either of the requirements, both measurements; Quasi-Peak and Average have to be carried out. Normally, there will be insufficient margin and PWB modifications will be necessary.

Note 5: If the RF-signal is a continuous sinusoidal wave, then the indications from either Peak, Quasi-Peak and Average-mode detectors will be equal.

2.6 Requirements

With conducted immunity measurements, the limit applicable to the product, e.g. draft IEC 801-6, can be used directly for verifying the IC application.

When radiated immunity requirements are given, a transformation from electric fieldstrength requirements to induced voltages and currents has to be considered. For indication the following relation can be considered:

Workbench EMC evaluation method

EIE/AN91001

- 1 Volt_{emf} (150 Ω in series) shall be taken for 1 Volt/metre (travelling wave, far field condition, generated by a tuned dipole antenna, distance ≥ 3 metre) in case the cables and EUT are exposed to the EM-field, i.e. transformation ratio = 0 dB.
- 0,3 Volt_{emf} (150 Ω in series) shall be taken for 1 Volt/metre (travelling wave, far field condition, generated by parallel plate or strip-line set-ups) in case only the EUT is exposed to the EM-field and cables are excluded, i.e. transformation ratio = -10 dB.

Note 6: The transformation ratio will depend on product size, the way cables are routed and frequency.

When carrying out conducted emission measurements in the frequency range, freq. ≥ 30 Mhz, where the radiation limits are given in μVolt/metre, measured at 10 metre distance, the following approximation can be applied:

$$E_{\text{limit}} = 30 \text{ dB}\mu\text{V/m} (= 30 \mu\text{V/m}) \text{ at 10 metre distance from the object.}$$

$$E = (7/d) \cdot \sqrt{P_t}, \quad P_t = U^2/150 \Omega,$$

$$E = (7/10) \cdot \sqrt{(U^2/150)} = U/17,5$$

$$E(\text{dB}\mu\text{V/m}) = U(\text{dB}\mu\text{V}) - 25 \text{ dB.}$$

In some emission standards, e.g. automotive and information technology equipment, conducted emission requirements are given over a large frequency range. Those requirements shall be adhered without any transformation.

In portable applications such as radio or television where an on-top antenna is used, functional requirements will be much more severe than the legal requirements. Those functional limits need to be measured on the product, without any annoying disturbances, and thereafter the relation given in chapter 2.6 can be applied to the found limit.

Workbench EMC evaluation method

Workbench EMC evaluation method EIE/AN91001

3 EXAMPLES.

All applications, e.g. a single op-amp, a transmission/speech circuit for telephone, μ P or video processor are tested on immunity with impedances (symmetrical and a-symmetrical loading) at inputs and outputs applied according to the application instructions (which shall be stated in the application report !!).

When these (input, output) impedances are passive, they can be either outside or inside the workbench Faraday cage. When these are active, they should be outside. When coaxial feed-throughs are used, care must be taken by additional measures (low-pass-filters, see Fig.5) that the RF-energy which will be superimposed on either input or output lines does not adversely affect the performance of the auxiliary equipment and that by measures the source or load impedance remains properly defined (even at RF).

The signal wires going from the application board to the feed-through connectors or filters shall be provided with an RF-blocking impedance, represented by 14 - 17 turns on a 4C65 (freq ≥ 1 MHz) or better 4A15 ferrite toroid.

By means of three 100 Ω resistors (PR 37 or PR 52, or equivalent power metal film resistors), the common of the input, output and supply is then coupled to the three coaxial feed-through connectors as indicated in paragraph 2.3. Externally, these connections are either coupled to the RF-disturbance generator, the selective voltmeter or terminated by a 50 Ω coaxial resistor.

NOTE 7: The lay-out of the PWB shall be made such that either the typical performance of the circuit is tested (non-optimized mono- or bi-layer) or on e.g. multi-layer with every precaution taken to have optimal EMC performance of the IC in this application.

NOTE 8: If the latter application cannot satisfy the EMC requirements, then no designer/customer will be capable of making a proper product with it at reasonable costs.

3.1 Audio applications.

3.1.1 Immunity

The audio circuit shall be set at nominal (gain) conditions. When necessary or obtainable, a 1 kHz generator can be used to make the required setting. The baseband 1 kHz output signal across the normal load shall be measured. This level will be taken as a reference for the immunity performance testing.

The disturbance signal (RF, 80 % modulated by 1 kHz) shall be applied to one of the common-ports of the PWB under test in turn, while the other ports are terminated to reference by 50 Ω . A demodulated signal level (1 kHz only) of 40 dB less than the nominal signal level across that load is acceptable for proper operation. This level can be measured either by a low frequency spectrum analyzer or a sensitive AC-Voltmeter with a 1 kHz band-pass in front (as described in EN 55020). A typical pass-band bandwidth of 500 Hz is sufficient for these kind of measurements.

Workbench EMC evaluation method

EIE/AN91001

Example: The nominal level on the a,b-lines of a telephone system is 100 mV (across 600 Ω). The signal level across the earpiece can be measured and may be 30 mV (example only !!, determined by dynamic sensitivity of the earpiece). The level of the demodulated signal across that same earpiece, with the disturbance signal applied to the PWB shall be less than 0.3 mV.

3.1.2 Emission

In most cases the emission from linear applications is nil unless it contains some oscillator. In the latter case, the disturbance generator shall be replaced by a spectrum analyzer covering the frequency range of interest, commonly 9 kHz to 1 GHz. As the emission will be continuous (at the fundamental and its harmonics), the detector chosen in the spectrum analyzer will not influence the readings. Where possible, the bandwidth requirements as stated in IEC CISPR publication 16 shall be adhered.

3.2 Video applications

3.2.1 Immunity

With video applications it will be such that most applications are based on an interlaced video frame sequence of 25 or 30 Hz. In both situations the spectrum around 1 kHz is fully crowded by harmonics of the frame frequency. It is therefore, that the modulation frequency, nominal 1 kHz should be shifted a little up/down such that it becomes an inter-harmonic of the frame frequency.

Examples: 25 Hz \rightarrow 987,5 Hz
30 Hz \rightarrow 1005 Hz

To obtain nominal settings of the video application a colour bar signal is applied to the circuit to be fully operational.

The demodulated signal at the outputs, e.g. CVBS, RGB or sync. can be measured by using a spectrum analyzer with a resolution bandwidth of ≤ 6 Hz. This selectivity is required to obtain a signal-to-noise ratio sufficient to discriminate the demodulated signal from the wanted signal. Here the demodulated signal to nominal signal level_(peak-to-peak) ratio has to be about -55 to -60 dB to be just not perceptible on the screen. This limit level can be found by superimposing a 1 kHz signal to the normal signal while observing the picture. The 1 kHz generator level is increased up to a level where it becomes just perceptible on the screen. This level referred to the functional signal is then taken as limit for the signal-to-interference ratio (S/I).

Note 9: Special care shall be taken to assure that the demodulation of the spectrum analyzer is much less than the limit level to be measured. Additional low-pass filters, see Fig.5, may be required !! Furthermore, it may be necessary to use a comb-filter to lower the level of the frame harmonics.

Example: The video signal level of the G-signal from the processor board to the video output stages is 3600 mV_{p-p}. Then the demodulated signal shall be less than 3,6 mV_(peak).

Note 10: Disturbing signals may also effect synchronization and as such appear as vertical zig-zag lines on the screen. This effect can only be measured by using a jitter or phase modulation meter between sync-out of the pattern generator and the output signal coming from the PWB.

Workbench EMC evaluation method

EIE/AN91001

3.2.2 Emission

As indicated above emission from the PWB is measured by replacing the disturbance generator by a spectrum analyzer or selective voltmeter. In analog applications, emission will be caused by the video signal itself and some oscillator signals used for demodulation, mixing, etc. According to the emission standard EN 55013, the test page pattern of teletext shall be used as video information. Where possible, the bandwidth and detector requirements as stated in IEC CISPR publication 16 shall be adhered.

3.3 Digital applications**3.3.1 Immunity**

With digital applications, the main problem will be the fact that it needs wide address and data buses for operation. As these wide buses are inconvenient for this method, these shall be limited by simplifying the circuit or by introducing parallel-to-series and series-to-parallel decoders. Preferable, these decoders must be used on the PWB under test such that only one serial line will be fed through the wall of the cage. To isolate the decoders from the IC under test, series resistors, e.g. 1 k Ω , shall be used in-between the decoders and the IC. The common-mode points on the PWB needs to be chosen close to the IC to be tested instead of the serial decoder input and output ports indicated in chapter 2.4.

By means of a tap, the signals at the output of the IC shall be monitored by means of an oscilloscope or a logic analyzer with an analog input and adjustable threshold levels. As criteria, the output signal of the IC may not exceed the specified high/low levels other than during functional transitions.

When testing the inputs, the worst case DC-levels shall be superimposed to the input signal (if it does not already contain a DC-component from a previous stage). When the disturbance signal is applied to the common-mode points chosen, the signal levels at the outputs shall be observed.

With more complex circuits it can be such that analog and digital inputs and outputs are available. By closing the loop; data \rightarrow dig.out \rightarrow dig.in \rightarrow analog out \rightarrow analog in \rightarrow compare with initial data and turn on/off a flag, which drives a LED, a powerful test program is carried out.

3.3.2 Emission

For proper operation, the normal program or coding shall be applied to the IC such that the emission measured is representative for a typical application. In case of a μ P, the outputs can be driven such that a digital ramp function (bit 0 = frequency F0, bit 1 = F0/2, etc) is generated over its 4, 8 or 16 bits wide data bus. All buses will have typical length e.g. 0,1 metre. The end of the bus shall be terminated as indicated in the application report e.g. by 50 pF/3.3 k Ω . The common-mode points for the IC under test shall be taken (similar as with immunity) to measure the emission performance of the IC in its application.

Workbench EMC evaluation method

EIE/AN91001

4 THE WORKBENCH CAGE PARAMETERS

The size of the workbench Faraday cage is chosen in such a way that it can contain most typical application and evaluation boards:

Length:	500 mm
Width :	350 mm
Height:	150 mm

The Workbench Faraday cage is made from carbon-free iron 1,5 mm thick. A conductive gasket is used between the box and the cover to make proper contact. The inside of the box is covered with an anti-static insulating material.

The connections through the wall can be made by:

Coax	:	5 x BNC,
Single line	:	4 x π -filter, (2 x 1,35 nF + 8 μ H, 2 Amp., 50 Volt max.),
	:	6 x Feed-through capacitors, (62 nF, 16 Amp., 500 Volt max.).

The shielding effectiveness in $H_{x,y,z}$ directions is better than 60 dB in the frequency range 1 to 1000 MHz, measured with two 60 mm electrically shielded loops according to Mil.Std. 220. An example of the shielding effectiveness of the workbench Faraday cage is given in Fig.7.

The characteristics of the π -filters (Low-Pass-Filter at 1 MHz) and the feed-through capacitors are given in Fig.8. The choice for these filters is made such that by additional external measures the filtering performance can be enhanced at low frequencies. For most applications the given performance will be sufficient. These π -filters can also be used at the outputs of digital circuits when using a 100 Ω resistor in series. In this case, the pass-band is limited to about 500 kHz, which is still sufficient to allow e.g. I²C-communication.

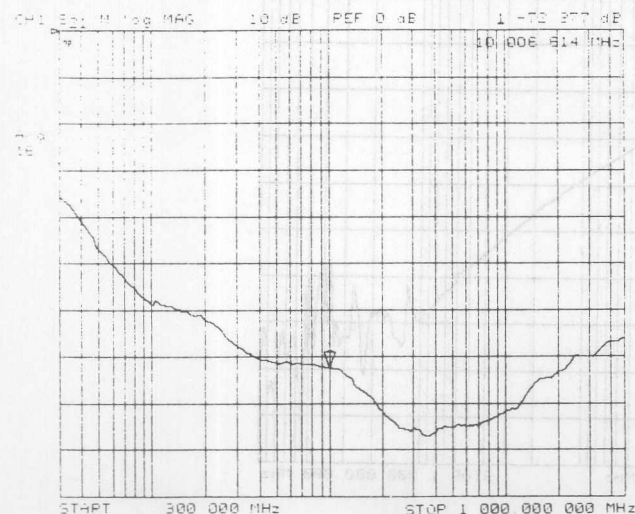


Fig.7. Shielding effectiveness of the workbench Faraday cage.

Workbench EMC evaluation method

EMC evaluation method IIE/AN91001

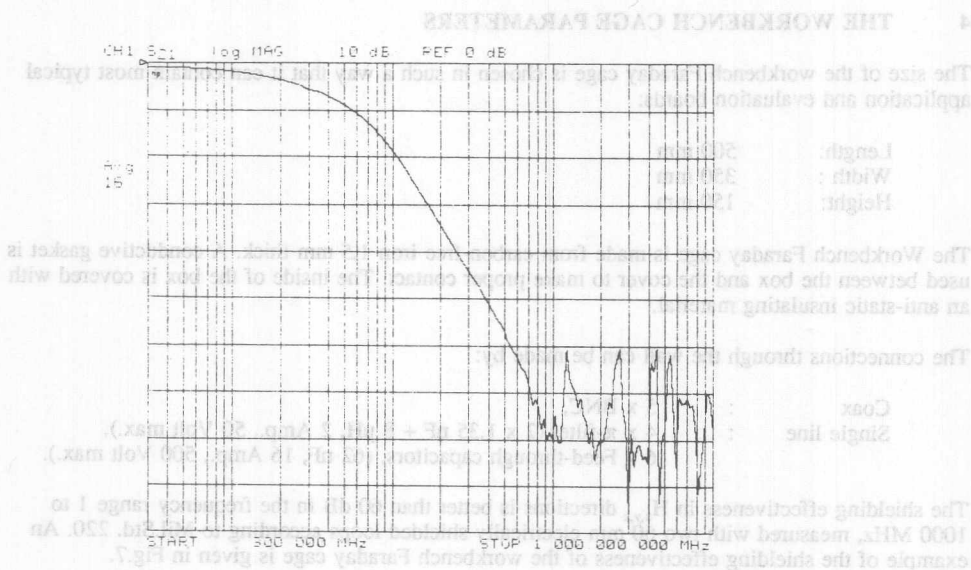


Fig.8a. Performance of the π -feed-through filter used with the workbench Faraday cage.

The choice for these filters is made such that by additional external measures the filtering performance can be enhanced at low frequencies. For most applications the given performance will be sufficient. These π -filters can also be used at the outputs of digital circuits when using a 100 Ω resistor in series. In this case the pass-band is limited to about 500 kHz, which is still sufficient to allow e.g. PC-compatibility.

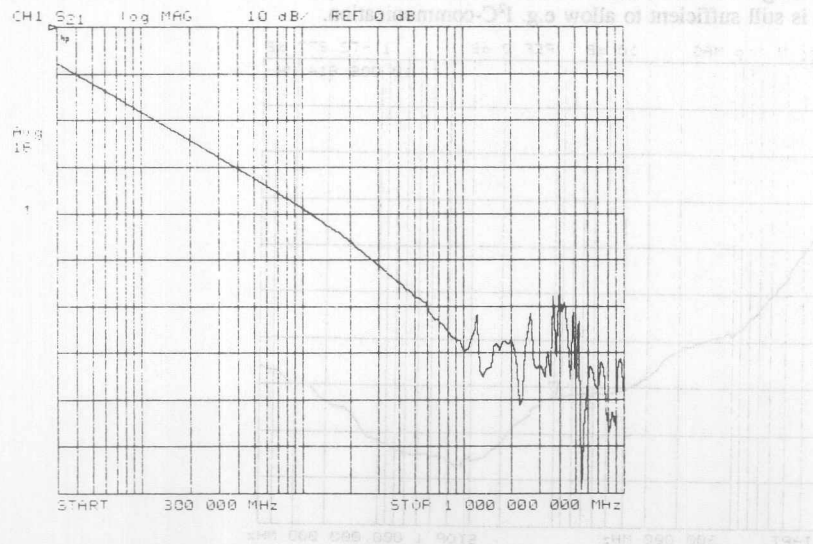


Fig.8b. Filter performance of the feed-through capacitor used with the workbench Faraday cage.

Workbench EMC evaluation method

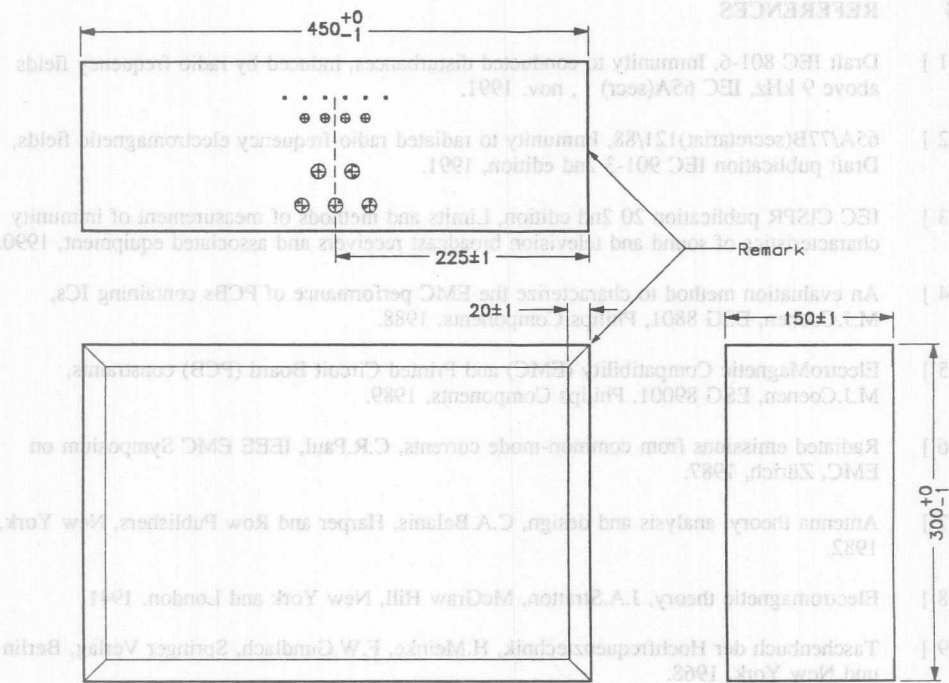
EIE/AN91001

5 REFERENCES

- [1] Drafts IEC 801-6, Immunity to conducted disturbances, induced by radio frequency fields above 9 kHz, IEC 65A(secr) , nov. 1991.
- [2] 65A/77B(secretariat)121/88, Immunity to radiated radio-frequency electromagnetic fields, Draft publication IEC 901-3 2nd edition, 1991.
- [3] IEC CISPR publication 20 2nd edition, Limits and methods of measurement of immunity characteristics of sound and television broadcast receivers and associated equipment, 1990.
- [4] An evaluation method to characterize the EMC performance of PCBs containing ICs, M.J.Coenen, ESG 8801, Philips Components, 1988.
- [5] ElectroMagnetic Compatibility (EMC) and Printed Circuit Board (PCB) constraints, M.J.Coenen, ESG 89001, Philips Components, 1989.
- [6] Radiated emissions from common-mode currents, C.R.Paul, IEEE EMC Symposium on EMC, Zürich, 1987.
- [7] Antenna theory, analysis and design, C.A.Balanis, Harper and Row Publishers, New York, 1982.
- [8] Electromagnetic theory, J.A.Stratton, McGraw Hill, New York and London, 1941.
- [9] Taschenbuch der Hochfrequenztechnik, H.Meinke, F.W.Gundlach, Springer Verlag, Berlin und New York, 1968.

Workbench EMC evaluation method

EIE/AN91001



Remark: Welded joints and finished

✓		UN-0 55	TOLERANCES UNLESS OTHERWISE STATED TOLERANTIES TENZIJ ANDERS VERMELD		UN-0 503			
R _a in µm			DIMENSION	ANGLE HOEK	UNIT		ASSEMBLY NO. SAMENSTELL.NOMMER.	QUANTITY MANTAL
			MAT		LITON STOK			
GENERAL EISENDE ALGEMENE VERBOD		UNIT EENHEID	CHROMIUM NICKEL STEEL PLATE 1 mm					
✓		mm						
SCALE SCHAL.		PROJ. EENW.	SHEAR-DRILL-FOLD-WELD-FINISH					
CLASS NO			EMC MEASURING TOOL					1 90-08-27
			Part: BOX					
NAME NAAM		DESIGN ONTW.	1-001					A3
A. Coolen								
DATE DAT.			90-08-20					
BY GEM.			N.Y. PHILIPS GLOEILAMPENFABRIEKEN Eindhoven - NEDERLAND					

Annex 1. Drawing of the workbench Faraday cage.

A/D conversion with P83CL410 PCF1252-x EIE/AN91006

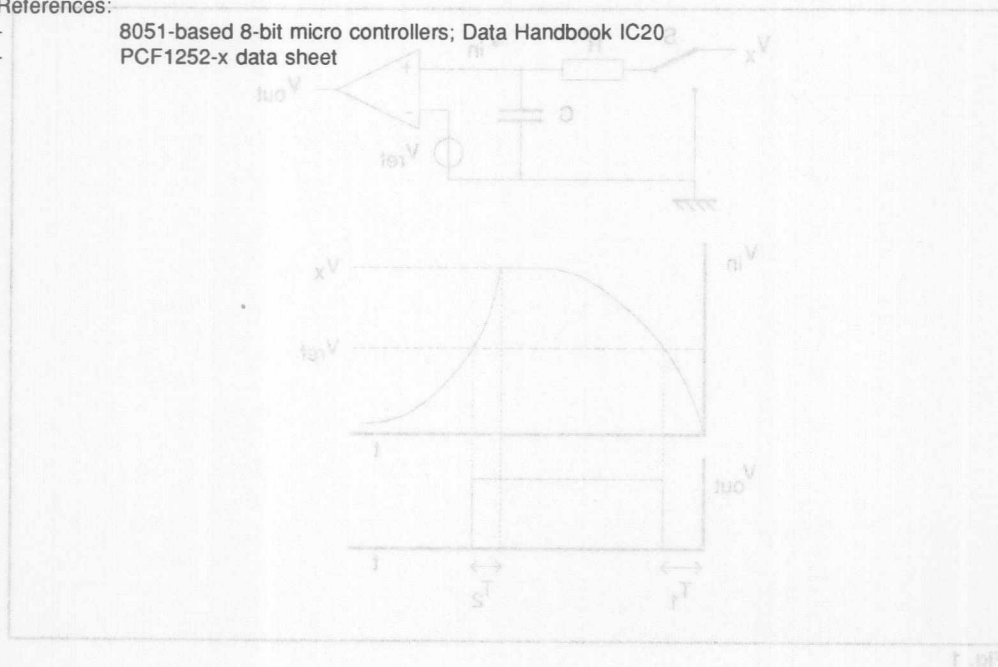
1. INTRODUCTION

With an 83CL410 micro controller and a PCF1252-x reset circuit it is possible to make a software driven A-to-D converter. In this application note an example is described where an 83CL410 measures its own supply voltage. The resolution of the measurement is 0.1V. The program example also refers to this application.

Chapter 2 describes the algorithm of the conversion. In chapter 3 the example with 83CL410/PCF1252 is described. Both hardware and software of this example are explained.

References:

- 8051-based 8-bit micro controllers; Data Handbook IC20
- PCF1252-x data sheet



Before the measurement is started, the analog switch connects the integrator input to ground, so that the integration capacitor is fully discharged. The measurement is started by connecting the integrator input to the unknown voltage. The integration capacitor will charge up. When the non-inverting input exceeds the reference voltage, the comparator output will switch from LOW to HIGH.

2. A-to-D conversion principle

The basic principle of the conversion is to convert the voltage to a time measurement. A micro controller without an on-chip A-to-D converter cannot measure voltages directly, but if converted to a time measurement, this can be done by software or with the help of an on-chip timer/count

Fig. 1 shows the basic circuit to do the conversion. The circuit consists of an integrator circuit, a voltage reference and an analog input switch S. The analog input switch is controlled by the microcontroller. The integrator is built around a comparator whose output is connected to a micro controller input.

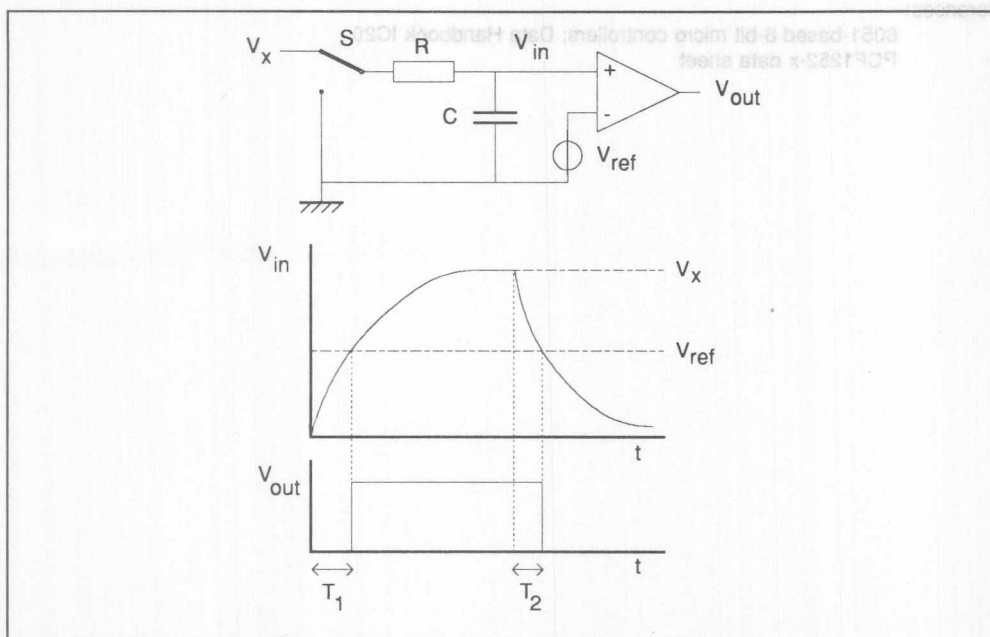


Fig. 1

Before the measurement is started, the analog switch connects the integrator input to ground, so that the integration capacitor is fully discharged. The measurement is started by connecting the integrator input to the unknown voltage. The integration capacitor will charge up. When the non-inverting input exceeds the reference voltage, the comparator output will switch from LOW to HIGH.

A/D conversion with P83CL410 PCF1252-x EIE/AN91006

3. EXAMPLE OF CONVERSION PRINCIPLE

The time between starting the measurement and the moment that the comparator output becomes HIGH, is measured by the microcontroller and is:

$$T_1 = -RC \ln \frac{V_x - V_{ref}}{V_x}$$

The charging continues until the capacitor is fully charged. Then the integrator input is grounded via the analog switch. The integration capacitor discharges while the comparator output is HIGH. When the input voltage becomes lower than the reference voltage, the comparator output becomes LOW again. The time between the start of the discharging and the moment that the comparator output becomes LOW, is measured by the microcontroller and is:

$$T_2 = -RC \ln \frac{V_{ref}}{V_x}$$

When the micro controller uses the ratio between T_1 and T_2 the result becomes independent of the values of R and C. The resulting ratio can then be used as a pointer to a look-up table that may contain an indication for the measured parameter or display data.

A/D conversion with P83CL410 PCF1252-x EIE/AN91006

3. EXAMPLE OF CONVERSION PRINCIPLE

3.1 Hardware

In the example an application is used where an microcontroller measures its own supply voltage. The supply voltage is generated by solar cells, so power consumption should be minimized. The measured voltage is shown on an LCD display.

An 83CL410 is used as microcontroller. This controller is a 80C51 family member. Compared with a standard 80C51 it has the following extra features:

- Wide supply voltage range of 1.8V to 6V.
- Wide operating frequency range of 32kHz to 20MHz with internal oscillator.
- With external oscillator there are no limitations on the lower frequency limit.
- Byte I²C interface instead of UART.
- 8 extra external interrupt inputs on P1. The interrupt level is programmable. These interrupts can terminate the power-down mode.
- 3 mask programmable I/O port configurations. These configurations are:
 - * Standard quasi-bidirectional I/O
 - * Open-drain output, standard input
 - * Push-pull output, no input
- I/O port levels after RESET are mask programmable.

Of these features only the byte I²C interface is not used.

The PCF1252 is used for monitoring the power supply voltage and generating a reset pulse when the supply drops too much. Several versions of PCF1252 are available, every one with its own trip voltage. The PCF1252 also has an unused comparator. This comparator is used for the integrator circuit. The inverting input is internally connected to a 1.3V reference.

Fig. 2 shows the A-to-D conversion part of the circuit.

R1 and C1 determine the time-constant of the circuit. The input of the integrator is connected to P1.1 of the micro controller. This port line has a push-pull configuration. It is used as the analog switch shown in fig. 1. When this output line is made HIGH, the integrator is connected to the supply voltage. This is the voltage to be measured. The voltages on the non-inverting input and the output are shown in fig.2. The output of the comparator is LOW, and the capacitor will be charged. When the voltage at the non-inverting input becomes higher than 1.3V, the comparator output will become HIGH. This HIGH level will cause an interrupt on P1.0 (IL2=1). The voltage on the inverting input will want to rise to $V_{cc}+1.3V$. This voltage is limited by 2 external diodes. R2 is a current limit resistor.

Next step is that P1.1 becomes LOW, so that the capacitor will be discharged. The P1.0 input is now programmed to generate an interrupt on a LOW level (IL2=0). This will happen when the voltage on the inverting input has dropped below 1.3V.

At this point both charge and discharge times are known, and the supply voltage can be calculated. In this circuit the PCF1252-9 is used, which will generate a reset pules when the supply voltage is lower than 2.55V. This limits the lower end of the supply voltage range. However, the 83CL410 is able to go as low as 1.8V.

A/D conversion with P83CL410 PCF1252-x EIE/AN91006

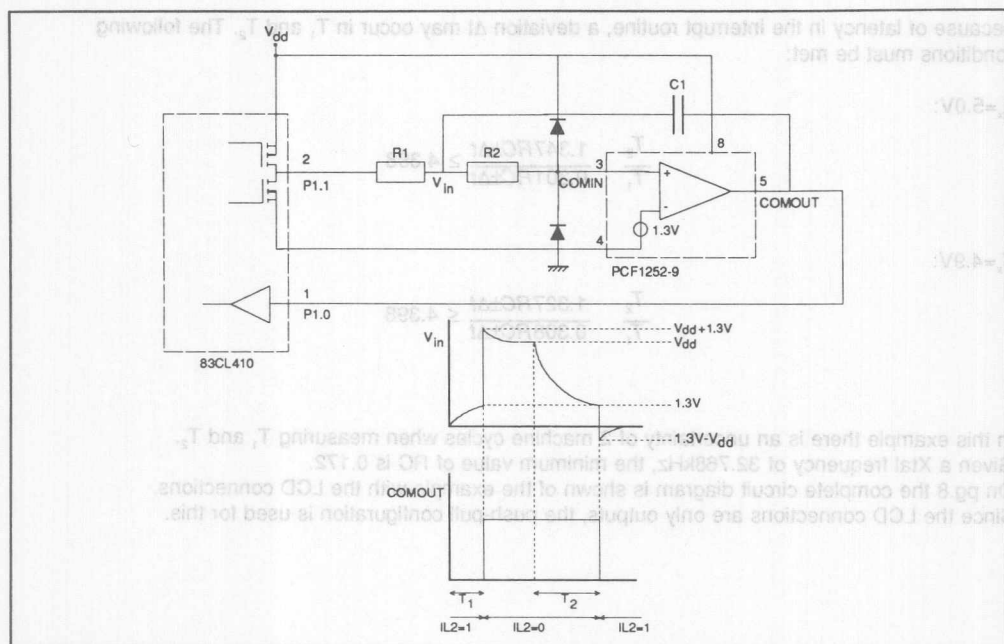


Fig. 2

To minimize power consumption, the lowest frequency is used where the internal oscillator can still be used. This is 32.768kHz. This parameter, together with the resolution that must be met, determine the minimum value of $R1 \cdot C1$. In this example a resolution of 0.1V is achieved. The software is made to handle voltage measurements from 1.8V to 5V.

To determine the minimum value of $R1 \cdot C1$, the measurement of 5.0V and 4.95V is important because here the distance between measurements of T_1 and T_2 is minimal.

$$\begin{aligned} V_x = 5.0V: & \quad T_1 = 0.301RC \\ & \quad T_2 = 1.347RC \\ & \quad T_2/T_1 = 4.437 \end{aligned}$$

$$\begin{aligned} V_x = 4.95V: & \quad T_1 = 0.304RC \\ & \quad T_2 = 1.337RC \\ & \quad T_2/T_1 = 4.398 \end{aligned}$$

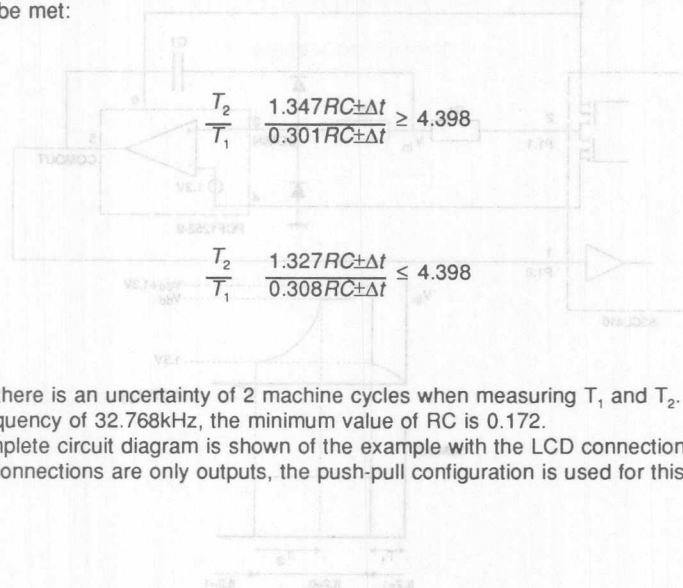
$$\begin{aligned} V_x = 4.9V: & \quad T_1 = 0.308RC \\ & \quad T_2 = 1.327RC \\ & \quad T_2/T_1 = 4.308 \end{aligned}$$

A/D conversion with P83CL410 PCF1252-x EIE/AN91006

Because of latency in the interrupt routine, a deviation Δt may occur in T_1 and T_2 . The following conditions must be met:

$V_x = 5.0V$:

$V_x = 4.9V$:



In this example there is an uncertainty of 2 machine cycles when measuring T_1 and T_2 .

Given a Xtal frequency of 32.768kHz, the minimum value of RC is 0.172.

On pg.8 the complete circuit diagram is shown of the example with the LCD connections.

Since the LCD connections are only outputs, the push-pull configuration is used for this.

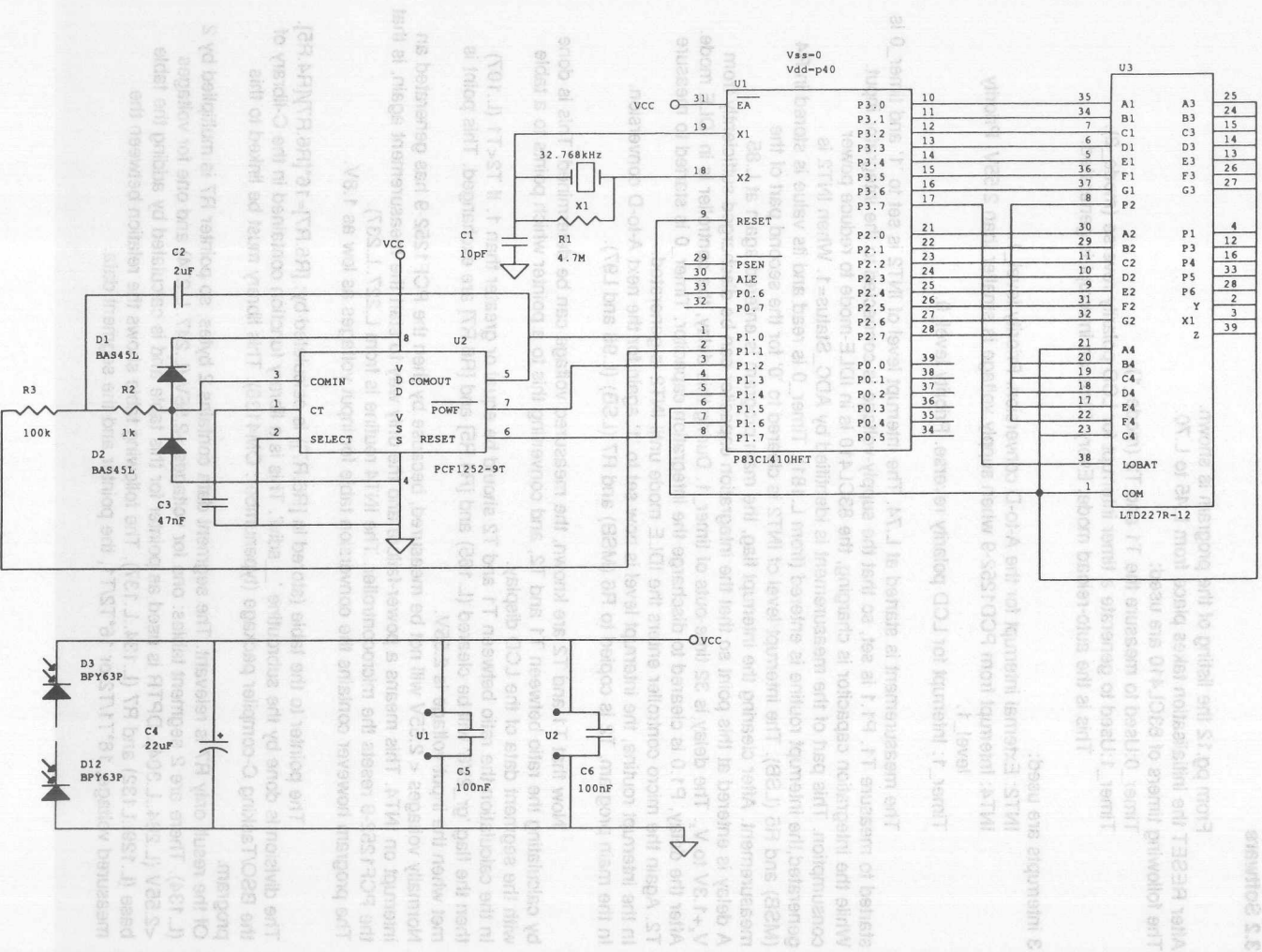
FIG. 2

To minimize power consumption, the lowest frequency is used where the internal oscillator can still be used. This is 32.768kHz. This parameter, together with the resolution that must be met, determine the minimum value of RC. In this example a resolution of 0.1V is required. The software is made to handle voltage measurements from 1.8V to 5V.

To determine the minimum value of RC, the measurement of 5.0V and 4.9V is important because here the distance between measurements of T_1 and T_2 is minimal.

$V_x = 5.0V$:	$T_1 = 0.301RC$
	$T_2 = 1.347RC$
	$T_2/T_1 = 4.47$
$V_x = 4.95V$:	$T_1 = 0.304RC$
	$T_2 = 1.337RC$
	$T_2/T_1 = 4.398$
$V_x = 4.9V$:	$T_1 = 0.308RC$
	$T_2 = 1.327RC$
	$T_2/T_1 = 4.308$

A/D conversion with P83CL410 PCF1252-X09 01AJ0389 niw EIE/AN91006



A/D conversion with P83CL410 PCF1252-x EIE/AN91006

3.2 Software

From pg.12 the listing of the program is shown.

After RESET the initialisation takes place from L.45 to L.70.

The following timers of 83CL410 are used:

Timer_0: Used to measure the T1 and T2 (mode_1).

Timer_1: Used to generate a timer interrupt for LCD polarity reverse (mode_2).

This is the auto-reload mode. Every 10ms an interrupt is generated.

3 interrupts are used:

INT2: External interrupt for the A-to-D conversion. Priority level_1.

INT4: Interrupt from PCD1252-9 when supply voltage is smaller than 2.55V. Priority level_1.

Timer_1: Interrupt for LCD polarity reverse. Priority level_0.

The measurement is started at L.74. The interrupt level of INT2 is set to '1' and timer_0 is started to measure T1. P1.1 is set, so that the supply-voltage is connected to the integrator input.

While the integration capacitor is charging, the 83CL410 is in IDLE-mode to reduce power consumption. This part of the measurement is identified by ADC_Status=1. When INT2 is generated, the interrupt routine is entered (from L.181). Timer_0 is read and its value is stored in R4 (MSB) and R5 (LSB). The interrupt level of INT2 is cleared to '0' for the second part of the measurement. After clearing the interrupt flag, the main program is entered again at L.85.

A delay is entered at this point so that the integration capacitor can be discharged sufficiently from $V_x + 1.3V$ to V_x . The delay is 32 time-outs of timer_1. During this delay, the controller is in IDLE mode. After the delay, P1.0 is cleared to discharge the integration capacitor. Timer_0 is started to measure T2. Again the micro controller enters the IDLE mode until INT2 is generated.

In the interrupt routine, the interrupt level is now set to '1' again for the next A-to-D conversion.

In the main program, T2 is copied to R6 (MSB) and R7 (LSB) (L.96 and L.97).

Now that T1 and T2 are known, the measured voltage can be determined. This is done by calculating the ratio between T1 and T2, and converting this to a pointer which points to a table with the segment data of the LCD display.

In the calculation the ratio between T1 and T2 should be equal or greater than 1. If $T2 < T1$ (L.107) then the flag 'gr_2V6' will be cleared (L.165) and [R4.R5] and [R6.R7] are exchanged. This point is met when the input voltage is 2.55V.

Normally voltages $< 2.55V$ will not be measured, because by then the PCF1252-9 has generated an interrupt on INT4. This means a power-failure and the only way to start the measurement again, is that the PCF1252-9 resets the microcontroller. The INT4 routine is from L.227..L.237).

The program however contains the conversion table for input voltages as low as 1.8V.

The pointer to the table (stored in [R6.R7]) is calculated by: $[R6.R7] = 16 * [R6.R7] / [R4.R5]$. The division is done by the subroutine '___sdivi'. This is a library function contained in the C-library of the BSO/Tasking C-compiler package (typenumber: OM4136). This library must be linked to this program.

Of the result only R7 is relevant. The segment data contains 2 bytes, so pointer R7 is multiplied by 2 (L.134). There are 2 segment tables: one for voltages $> 2.55V$ (L.247..L.274) and one for voltages $< 2.55V$ (L.284..L.304). DPTR is used as pointer for this table and is calculated by adding the table base (L.129..L.132) and R7 (L.134..L.137). The following table shows the relation between the measured voltage, $16 * T1/T2$ or $16 * T2/T1$, the pointer and the segment data.

A/D conversion with P83CL410 PCF1252-x EIE/AN91006

Input voltage >2.55V				
Input Voltage	16*T2/T1	Table addresses	Segment data	Display
4.95	70.4	140..147	0x6d,0xbf	5.0 V
4.85	67.5	136..139	0x66,0xef	4.9 V
4.75	64.83	130..135	0x66,0xff	4.8 V
4.65	62.18	124..129	0x66,0x87	4.7 V
4.55	59.57	120..123	0x66,0xfd	4.6 V
4.45	56.98	114..119	0x66,0xed	4.5 V
4.35	54.43	108..113	0x66,0xeb	4.4 V
4.25	51.91	104..107	0x66,0xcf	4.3 V
4.15	49.42	98..103	0x66,0xdb	4.2 V
4.05	46.96	94..97	0x66,0x86	4.1 V
3.95	44.54	90..93	0x66,0xbf	4.0 V
3.85	42.14	84..89	0x4f,0xef	3.9 V
3.75	39.82	80..83	0x4f,0xff	3.8 V
3.65	37.51	76..79	0x4f,0x87	3.7 V
3.55	35.24	70..75	0x4f,0xfd	3.6 V
3.45	33.02	66..69	0x4f,0xed	3.5 V
3.35	30.83	62..65	0x4f,0xeb	3.4 V
3.25	28.7	58..61	0x4f,0xcf	3.3 V
3.15	26.6	54..57	0x4f,0xdb	3.2 V
3.05	24.56	48..53	0x4f,0x86	3.1 V
2.95	22.56	46..47	0x4f,0xbf	3.0 V
2.85	20.62	42..45	0xdb,0xef	2.9 V LOBAT
2.75	18.73	38..41	0xdb,0xff	2.8 V LOBAT
2.65	16.89	34..37	0xdb,0x87	2.7 V LOBAT
2.55	15.12	32..34	0xdb,0xfd	2.6 V LOBAT

A/D conversion with P83CL410 PCF1252-x EIE/AN91006

Input voltage <2.55V				
Input voltage	16*T1/T2	Table addresses	Segment data	Display
2.55	16.92	32..37	0xdb,0xed	2.5 V LOBAT
2.45	19.09	38..43	0xdb,0xeb	2.4 V LOBAT
2.35	21.77	44..49	0xdb,0xcf	2.3 V LOBAT
2.25	25.15	50..57	0xdb,0xdb	2.2 V LOBAT
2.15	29.51	58..69	0xdb,0x86	2.1 V LOBAT
2.05	35.32	70..85	0xdb,0xbf	2.0 V LOBAT
1.95	43.35	86..109	0x86,0xef	1.9 V LOBAT
1.85	54.96	110..119	0x86,0xff	1.8 V LOBAT

Now the data can be displayed on the LCD display. When writing data to the LCD, the timer_1 interrupt is disabled. The value of output LCD_COM (controlled by timer_1 interrupt routine) determines whether the data must be written inverted to the LCD or not. The data is written non-inverted from L.147..L.150, inverted from L.153..L.158. When the data is written, timer_1 interrupt is enabled again, and a new A-to-D conversion is started at label Start_ADC.

From L.210..L.219 the timer_1 interrupt routine is shown. In this routine the segment lines to the LCD display are inverted (on P2 and P3). Also LCD_COM is inverted, which is the COMMON pin of the display.

On L.218 the value Djs_tim is incremented. This is for the delay between the measurement of T1 and T2 (see L.85).

2.5 V LOBAT	0xdb,0xed	32..37	16.92	2.55
2.4 V LOBAT	0xdb,0xeb	38..43	19.09	2.45
2.3 V LOBAT	0xdb,0xcf	44..49	21.77	2.35
2.2 V LOBAT	0xdb,0xdb	50..57	25.15	2.25
2.1 V LOBAT	0xdb,0x86	58..69	29.51	2.15
2.0 V LOBAT	0xdb,0xbf	70..85	35.32	2.05
1.9 V LOBAT	0x86,0xef	86..109	43.35	1.95
1.8 V LOBAT	0x86,0xff	110..119	54.96	1.85

A/D conversion with P83CL410 PCF1252-x EIE/AN91006

4: SOFTWARE LISTING

TSW-ASM51 V3.0b Serial #00052252 Main program
PAGE 1

```

LOC   OBJ          LINE   SOURCE
*****
1     $TITLE(Main program)
2     $DEBUG
3     $
4     # 1 "C:\Tools\Tasking\ASM51\Include\8051\Reg410.Inc"
5
;*****
6     ;Timer_0 is used for A/D conversion in combination with PCF1252
7     ;Timer_1 is used as time base for LCD switching
8     ;INT2 is used as input for A/D conversion
9     ;INT4 is used as power-failure input from PCF1252
10    ;All timing related to Xtal=32.768kHz
11
;*****
12
0091  13             Anal_in   EQU   91h      ;P1.1 is voltage switch
0095  14             LCD_COM   EQU   95h      ;P1.5 is COM of LCD display
00B0  15             Digit_1   EQU   0B0h     ;P3 is first digit of LCD
                                           ;display
00A0  16             Digit_2   EQU   0A0h     ;P2 is second digit of LCD
                                           ;display
17
18
;*****
19
20
21             Flags   segment Bit
22             Stack   segment Data
23             Main    segment Code Inblock
24             Table_1 segment Code Page
25             Table_2 segment Code Page
26             ADC      segment Code
27             LCD       segment Code
28             Power_F  segment Code
29
----
30             RSEG Flags
0000:   R 31  ADC_Status:  DBIT 1      ;Status flag AD converter
0001:   32  Gr_2v6:       DBIT 1      ;Flag indicating that V<2.55V
33
----
34             RSEG Stack
0000:   R 35  Stk_St:     DS 20       ;Define stack
36
REG END 37             Dis_tim set r2 ;Timer for cap. discharge from Vdd+1.3
                                           ;to Vdd
38
39

```

A/D conversion with P83CL410 PCF1252-x EIE/AN91006

TSW-ASM51 V3.0b Serial #00052252 Main program

PAGE 2

LOC OBJ LINE SOURCE

;*****

```

40
---- 41          CSEG AT 00          ;Reset vector
0000: 020000 R 42          ljmp Start
43
---- 44          RSEG Main
0000: C291 R 45 Start:          clr Anal_in          ;Ground analog input
0002: 7581FF R 46          mov sp,#Stk_St-1 ;Initialise stack pointer
0005: C200 R 47          clr ADC_Status ;Initialise ADC status
48
0007: 75F805 49          mov ipl,#05h ;INT2 is priority level 1 (Analog
                                ;measurement.)
50          ;INT4 is priority level 1 (Power fail)
51          ;Timer_1 interrupt level 0 (LCD switch)
000A: 53E9FB 52          anl ixl,#0fbh ;INT4 on low level
000D: D2EA 53          setb ex4 ;Enable INT4
000F: D2AF 54          setb ea ;Global enable interrupts
55
0011: 758921 56          mov tmod,#21h ;Timer_1 mode (LCD switch)
57          ; 8 bit auto reload mode
58          ;Timer_0 mode (A/D conversion)
59          ; 16 bit timer
0014: 758DF5 60          mov t1l,#0f5h ;Delay for discharging cap's (+/- 1sec)
0017: 758B50 61          mov t1l,#050h
001A: D28E 62          setb tr1
001C: 308FFD 63          jnb tf1,$
001F: C28F 64          clr tf1
0021: C28E 65          clr tr1
66
0023: 758BE1 67          mov t1l,#0elh ;LCD switch rate: 10ms
0026: 758DE1 68          mov th1,#0elh
0029: D28E 69          setb tr1 ;Start timer_1
002B: D2AB 70          setb et1 ;Enable timer_1 interrupt
71
72 ;Measurement of supply voltage
002D: 43E901 73          orl ixl,#01H ;INT2 on '1' input level
0030: 74 Start_ADC:
0030: 758CFF 75          mov th0,#0FFh ;Load timer_0
0033: 758AFB 76          mov t10,#0FBh
0036: D28C 77          setb tr0 ;Start timer_0
0038: D291 78          setb Anal_in ;Start raising part of measurement
003A: D2E8 79          setb ex2 ;Enable INT2
003C: 438701 80 Id_1:          orl pcon,#01 ;Go to idle mode
003F: 3000FA R 81          jnb ADC_Status,Id_1;Wait till first part of measurement
                                ;is handled
82          ;Exit from idle mode could be from LCD
                                ;interrupt

```

A/D conversion with P83CL410 PCF1252-x EIE/AN91006

TSW-ASM51 V3.0b Serial #00052252 Main program

PAGE 3

LOC	OBJ	LINE	SOURCE
0042:	7A00	83	mov Dis_tim,#00 ;Discharge delay: Vdd+1.3...Vdd
0044:	438701	84	Id_2: orl pcon,#01 ;Go to idle mode
0047:	BA20FA	85	cjne Dis_tim,#20h,Id_2 ;Delay: 32 LCD_time_outs
		86	;If not equal: idle
		87	
004A:	758CFF	88	mov th0,#0FFh ;Load timer_0
004D:	758AFB	89	mov tl0,#0FBh
0050:	D28C	90	setb tr0 ;Start timer_0
0052:	C291	91	clr Anal_in ;Start second part of measurement is handled
		92	
0054:	438701	93	Id_3: orl pcon,#01 ;Go to idle mode
0057:	2000FA	R 94	jb ADC_Status,Id_3 ;Wait till measurement is finished
005A:	C2E8	95	clr ex2 ;Disable INT2
005C:	AE8C	96	mov r6,th0 ;Get timer data of second part of measurement
005E:	AF8A	97	mov r7,tl0 ;T2
		98	;Process data
		99	
		100	;If T1>T2 then [R6.R7] and [R4.R5] must
		101	;be exchanged. Also other segment table
		102	;must be used.
0060:	D201	R 103	setb gr_2v6 ;Set gr_2v6 flag
0062:	C3	104	clr c ;Test T2-T1
0063:	EE	105	mov a,r6
0064:	9C	106	subb a,r4
0065:	404C	107	jc Small_2v6 ;<2.6V: clear flag, exchange registers
0067:	7004	108	jnz Calculate
0069:	EF	109	mov a,r7
006A:	9D	110	subb a,r5
006B:	4046	111	jc Small_2v6
		112	;Calculate T2/T1
		113	;[R6.R7]=[R6.R7]/[R4.R5]
		114	;1<Result<4.4
		115	;Before dividing: [R6.R7]=16*[R6.R7]
006D:	7806	116	Calculate: mov r0,#06 ;r0 points to r6
006F:	EF	117	mov a,r7
0070:	C4	118	swap a
0071:	54F0	119	anl a,#0f0h
0073:	CF	120	xch a,r7 ;r7 shifted 4 bits left
0074:	C4	121	swap a
0075:	CE	122	xch a,r6
0076:	C4	123	swap a ;nibbles r6 swapped
0077:	D6	124	xchd a,@r0
0078:	FE	125	mov r6,a ;r6 also shifted 4 bits to the left
		126	

A/D conversion with P83CL410 PCF1252-x EIE/AN91006

TSW-ASM51 V3.0b Serial #00052252 Main program

PAGE 4

LOC	OBJ	LINE	SOURCE
0079:	120000	R 127	lcall __sdivi ;[R6.R7]=16*T2/T1
		128	
007C:	300105	R 129	jnb Gr_2v6,Sec_tab
007F:	900000	R 130	mov dptr,#Seg_tab_1
0082:	8003	131	sjmp Calc_point
0084:	900000	R 132	Sec_tab: mov dptr,#Seg_tab_2
0087:	C3	133	Calc_point: clr c
0088:	EF	134	mov a,r7 ;Calculate address of segment info
0089:	33	135	rlc a ;2*R7
008A:	F582	136	mov dpl,a
008C:	C3	137	clr c
008D:	9494	138	subb a,#148 ;If a>148 then V>5.0: Display H.I.V
008F:	4003	139	jc Display
0091:	758294	140	mov dpl,#148 ;V>5.0V
		141	
0094:		142	Display: ;Display result. DPTR points to segment
	data		
0094:	C2AB	143	clr etl ;Disable LCD switch interrupt
0096:	E4	144	clr a
0097:	93	145	movc a,@a+dptr ;Get segment_1 data (digit + LOBAT)
0098:	209509	146	jb LCD_COM,Com_1 ;If LCD_COM=1, then invert result
009B:	F5B0	147	mov Digit_1,a
009D:	7401	148	mov a,#01
009F:	93	149	movc a,@a+dptr ;Get segment_2 data (digit + point)
00A0:	F5A0	150	mov Digit_2,a
00A2:	01AF	R 151	ajmp New_meas
00A4:		152	Com_1:
00A4:	64FF	153	xrl a,#0ffh
00A6:	F5B0	154	mov Digit_1,a
00A8:	7401	155	mov a,#01
00AA:	93	156	movc a,@a+dptr ;Get segment_2 data (digit+point)
00AB:	64FF	157	xrl a,#0ffh
00AD:	F5A0	158	mov Digit_2,a
00AF:		159	New_meas:
00AF:	D2AB	160	setb etl ;Enable LCD switch interrupt
00B1:	0130	R 161	ajmp Start_ADC
		162	
		163	
00B3:		164	Small_2v6: ;V<2.55V. Exchange registers
00B3:	C201	R 165	clr Gr_2v6 ;Clear flag
00B5:	CC	166	xch a,r4 ;Exchange [R4.R5] and [R6.R7]
00B6:	CE	167	xch a,r6
00B7:	FC	168	mov r4,a
00B8:	CD	169	xch a,r5
00B9:	CF	170	xch a,r7
00BA:	FD	171	mov r5,a
00BB:	80B0	172	sjmp Calculate ;Start calculation

A/D conversion with P83CL410 PCF1252-x EIE/AN91006

TSW-ASM51 V3.0b Serial #00052252 Main program

PAGE 5

```

LOC  OBJ          LINE  SOURCE
173
174      extrn code(,sdiv1)
175      $EJECT
176
177
;*****
178      ;A/D measurement interrupt routine (INT2)
179
;*****
180
----      181      CSEG AT 3Bh ;INT2 vector (Voltage measurement)
003B: 020000  R  182      ljmp ADC_Int
183      CSEG AT 04Bh
----      184      RSEG ADC
0000: C28C  R  185      ADC_Int:  clr tr0 ;Stop timer_0
0002: B200  R  186      cpl ADC_Status
0004: 300009  R  187      jnb ADC_Status,Stat_0
188
189      ;ADC_Status is '1'
0007: AC8C  190      mov r4,th0 ;Get timer_0 values
0009: AD8A  191      mov r5,t10 ;T1
000B: 53E9FE  192      anl ix1,#0FEh ;INT2 on '0' input level
193
194      ;Check discharge Vd+1.3V
195
000E: 8003  196      sjmp Leave_INT2
197
198      ;ADC_Status is '0'
0010: 43E901  199      Stat_0:  orl ix1,#01H ;INT2 on '1' input level
200
0013: C2C0  201      Leave_INT2:  clr iq2 ;Clear interrupt flag
0015: 32  202      reti
203
204
205      $seject
206
207
;*****
208      ;Timer_1 interrupt for inverting LCD signals
209
;*****
210
----      211      CSEG AT 1Bh ;Timer_1 interrupt (LCD switch)
001B: 020000  R  212      ljmp LCD_int
213
----      214      RSEG LCD
0000: 63B0FF  R  215      LCD_int:  xrl Digit_1,#0ffh ;Invert digit_1
0003: B295  216      cpl LCD_COM ;Invert LCD_COM
0005: 63A0FF  217      xrl Digit_2,#0ffh ;Invert digit_2

```

```

LOC  OBJ      LINE  SOURCE
0008: 0A      218      (iv)inc Dis_timer ;Update discharge timer
0009: 32      219      reti
                220
                221 $eject
                222
                223 ;*****
;*****
                224 ;Power failure interrupt routine (INT4)
                225
;*****
                226
----
                227      CSEG AT 04Bh ;Power failure interrupt from PCF1252
004B: 020000  R  228      ljmp P_Fail
                229      0_xomif qpcq:
----
                230      RSEG Power_F
0000: C291  R  231      P_Fail: 0_ja;e clr Anal_in; ;Ground analog input
0002: E4      232      clr a
0003: F5B0      233      mov Digit_1,a ;Clear LCD
0005: C295      234      clr LCD_COM
0007: F5A0      235      mov Digit_2,a
0009: 80FE      236      sjmp $,Int ;Exit only with RESET !!
                237
                238 ;$eject
                239
                240

```


A/D conversion with P83CL410 PCF1252-x EIE/AN91006

```

TSW-ASM51 V3.0b Serial #00052252 Main program
PAGE 7
LOC OBJ LINE SOURCE
241 ;Table with LCD segment data for V>=2.55V
242
243
---- 244 rseg table_1
245
0000: R 246 Seg_Tab_1: ;Table with LCD segment data for V>=2.55V
0000: 247 ds 32 ;V<2.55V. (Adr:0..31)
0020: DBFD 248 db 0dbh,0fdh ;'2.6V LOBAT'
; (Adr:32..33)
0022: DB87DB87 249 db 0dbh,87h,0dbh,87h ;'2.7V LOBAT'
; (Adr:34..37)
0026: DBFFDBFF 250 db 0dbh,0ffh,0dbh,0ffh ;'2.8V LOBAT'
; (Adr:38..41)
002A: DBEFDBEF 251 db 0dbh,0efh,0dbh,0efh ;'2.9V LOBAT'
; (Adr:42..45)
002E: 4FBF 252 db 04fh,0bfh ;'3.0V' (Adr:46..47)
0030: 4F864F86 253 db 04fh,86h,04fh,86h,04fh,86h ;'3.1V' (Adr:48..53)
0034: 4F86 254 db 4fh,0dbh,4fh,0dbh ;'3.2V' (Adr:54..57)
0036: 4FDB4FDB 255 db 4fh,0cfh,4fh,0cfh ;'3.3V' (Adr:58..61)
003A: 4FCF4FCF 256 db 4fh,0e6h,4fh,0e6h ;'3.4V' (Adr:62..65)
003E: 4FE64FE6 257 db 4fh,0edh,4fh,0edh ;'3.5V' (Adr:66..69)
0042: 4FED4FED 258 db 4fh,0fdh,4fh,0fdh,4fh,0fdh ;'3.6V' (Adr:70..75)
0046: 4FFD 259 db 4fh,87h,4fh,87h ;'3.7V' (Adr:76..79)
004A: 4FFD 260 db 4fh,0ffh,4fh,0ffh ;'3.8V' (Adr:80..83)
004E: 4FEF4FEF 261 db 4fh,0efh,4fh,0efh,4fh,0efh ;'3.9V' (Adr:84..89)
0052: 4FEF 262 db 66h,0bfh,66h,0bfh ;'4.0V' (Adr:90..93)
0056: 66BF66BF 263 db 66h,86h,66h,86h ;'4.1V' (Adr:94..97)
005A: 66866686 264 db 66h,0dbh,66h,0dbh,66h,0dbh ;'4.2V' (Adr:98..103)
005E: 66DB66DB 265 db 66h,0cfh,66h,0cfh ;'4.3V' (Adr:104..107)
0062: 66CF66CF 266 db 66h,0e6h,66h,0e6h,66h,0e6h ;'4.4V' (Adr:108..113)
0066: 66E6 267 db 66h,0edh,66h,0edh,66h,0edh ;'4.5V' (Adr:114..119)
006A: 66ED66ED 268 db 66h,0fdh,66h,0fdh ;'4.6V' (Adr:120..123)
006E: 66FD66FD 269 db 66h,87h,66h,87h,66h,87h ;'4.7V' (Adr:124..129)
0072: 66876687 270 db 66h,0ffh,66h,0ffh,66h,0ffh ;'4.8V' (Adr:130..135)
0076: 66FF 271 db 66h,0efh,66h,0efh ;'4.9V' (Adr:136..139)
007A: 66EF66EF 272 db 06dh,0bfh,06dh,0bfh,06dh,0bfh ;'5.0V' (Adr:140..147)
007E: 6DBF6DBF 273 db 06dh,0bfh
0082: 6DBF 274 db 76h,0b0h ;'H.IV' (Adr:148..149)
0086: 76B0 275
008A: 276 $EJECT

```

A/D conversion with P83CL410 PCF1252-x EIE/AN91006

```

TSW-ASM51 V3.0b Serial #00052252 Main program
PAGE      8

LOC  OBJ          LINE  SOURCE
-----
277
278
;*****
279 ;Table with LCD segment data for V<=2.55V
280
;*****
281
----
282      rseg table_2
283
0000:  R 284  Seg_Tab_2:
0000:      ds 32 ;V>2.55V. (Adr:0..31)
0020: DBEDDBED 286      db 0dbh,0edh,0dbh,0edh,0dbh,0edh;'2.5V LOBAT'
0024: DBED 287      ;(Adr:32..37)
0026: DBE6DBE6 287      db 0dbh,0e6h,0dbh,0e6h,0dbh,0e6h;'2.4V LOBAT'
002A: DBE6 287      ;(Adr:38..43)
002C: DBCFDBCF 288      db 0dbh,0cfh,0dbh,0cfh,0dbh,0cfh;'2.3V LOBAT'
0030: DBCF 288      ;(Adr:44..49)
0032: DBDBDBDB 289      db 0dbh,0dbh,0dbh,0dbh,0dbh,0dbh;'2.2V LOBAT'
0036: DBDB 289      ;(Adr:50..57)
0038: DBDB 290      db 0dbh,0dbh
003A: DB86DB86 291      db 0dbh,86h,0dbh,86h,0dbh,86h;'2.1V LOBAT'
003E: DB86 291      ;(Adr:58..69)
0040: DB86DB86 292      db 0dbh,86h,0dbh,86h,0dbh,86h
0044: DB86 292      ;(Adr:70..85)
0046: DBBFBDBF 293      db 0dbh,0bfh,0dbh,0bfh,0dbh,0bfh;'2.0V LOBAT'
004A: DBBF 293      ;(Adr:70..85)
004C: DBBFBDBF 294      db 0dbh,0bfh,0dbh,0bfh,0dbh,0bfh
0050: DBBF 294      ;(Adr:86..109)
0052: DBBFBDBF 295      db 0dbh,0bfh,0dbh,0bfh
0056: 86EF86EF 296      db 86h,0efh,86h,0efh,86h,0efh;'1.9V LOBAT'
005A: 86EF 296      ;(Adr:86..109)
005C: 86EF86EF 297      db 86h,0efh,86h,0efh,86h,0efh
0060: 86EF 297      ;(Adr:110..121)
0062: 86EF86EF 298      db 86h,0efh,86h,0efh,86h,0efh
0066: 86EF 298      ;(Adr:122..133)
0068: 86EF86EF 299      db 86h,0efh,86h,0efh,86h,0efh
006C: 86EF 299      ;(Adr:134..145)
006E: 86FF86FF 300      db 86h,0ffh,86h,0ffh,86h,0ffh;'1.8V LOBAT'
0072: 86FF 300      ;(Adr:146..157)
0074: 86FF86FF 301      db 86h,0ffh,86h,0ffh,86h,0ffh
0078: 86FF 301      ;(Adr:158..169)
007A: 38BF38BF 302      db 38h,0bfh,38h,0bfh,38h,0bfh;'1.7V LOBAT'
007E: 38BF 302      ;(Adr:170..181)
0080: 38BF38BF 303      db 38h,0bfh,38h,0bfh,38h,0bfh
0084: 38BF 303      ;(Adr:182..193)
304
0086: 305      end

```

Driver for 8xC851 E2PROM

EIE/AN91009

1: INTRODUCTION.

A set of software functions is given to access the E²PROM on 8xC851 μ C's. These functions can be called from application programs written in assembly, PL/M-51 or C. The functions are found in the E2PROM.OBJ file that can be linked to the application program.

The driver is written and tested with the following software tools from BSO-Tasking:

Assembler: ASM51 V3.2 (OM4142)

PL/M51: PL/M51 V3.0A (OM4144)

C Comp.: C51 V2.0 (OM4136)

Debugger: XRAY51 V1.4c (OM4129)

Resources used by driver:

Exclusive use of 1 register bank (default RB1)

Accumulator

PSW

1 static bit addressable RAM byte

Driver for 8xC851 E2PROM

EIE/AN91009

2: FUNCTION DESCRIPTIONS

The functions that use write and/or erase actions are interrupt driven except for E2PROM_wr_byte_pol. The application can check the status of these actions by testing the flag E2PROM_BUSY. This flag is available via the function E2PROM_status.

In the 8xC851, the E²PROM interrupt is combined with the UART interrupt. To enable the E²PROM interrupt, EA (in the IE-register) must be set (should be done in application program), the combined UART/E²PROM enable bit must be set (ES in the IE-register, done with function E2PROM_int_en) and the E²PROM interrupt enable bit (EEINT in ECNTRL register) must be set. The E²PROM interrupt flag is automatically set by functions that use erase/write actions. This means that the UART interrupt enable can not be disabled while the E²PROM interrupt is completely enabled. The E²PROM can be disabled separately with the E2PROM_int_dis function.

The priority level for UART and E²PROM interrupt are the same and are defined with the E2PROM_int_en function.

The E²PROM driver has a link to a UART interrupt handler. When an UART interrupt occurs, the status of the controller is pushed on the stack and then interrupt flags are tested to determine the source of the interrupt. When the source of the interrupt is the UART, then subroutine _UART_HDL is called. The implementation of the UART interrupt handler is done by the user. On the disk a file UART.SRC is available that contains this subroutine. This routine will only clear the trx-interrupt flag (TI) and rcv-interrupt flag (RI).

Driver for 8xC851 E2PROM

EIE/AN91009

2.1 E2PROM_init

Function description:

This function must be called before any of the other functions is called. The timing register for writing/erasing the E²PROM is initialised and the register bank that the E²PROM functions can use is defined.

The default registerbank is RB1; the ETIM register which determines the write/erase timing, is default initialised with 0x7B (Xtal = 12MHz). If other values are required, the parameters REGISTERBANK and XTAL must be changed in the equate list of the source file (E2PROM.ASM).

The E²PROM/UART interrupt is enabled and set to priority level '0'.

Calling Sequence:

```
E2PROM_init();
```

Function prototype:

```
void E2PROM_init (void)
```

Parameters:

None

Driver for 8xC851 E2PROM

EIE/AN91009

2.2 E2PROM_int_en

Function description:

This function will enable the E²PROM/UART interrupt. The global enable bit EA is not effected and must be controlled by the application program.

The priority level of the E²PROM/UART is controlled by the parameter 'Pr_Level'.

Calling sequence:

E2PROM_int_en (Pr_Level);

Function prototype:

void E2PROM_int_en (data char Pr_Level)

Parameters:

Pr_Level: This parameter determines the priority level on which the E²PROM/UART interrupts are handled. Values greater than 0x01 will be interpreted as 0x01.

2.3 E2PROM_int_dis

Function description:

This function will disable the E2PROM interrupt.

Calling sequence:

E2PROM_Int_Dis;

Function prototype:

void E2PROM_Int_Dis (void)

Parameters:

None

Driver for 8xC851 E2PROM

M09932 I2S EIE/AN91009

2.4 E2PROM_status

Function description:

This function will return the E2PROM_BUSY bit, which indicates whether a read/write transfer from/to the E²PROM, or an erase action is finished.

'1' indicates that a read/write transfer is still in progress.

'0' indicates that no read/write transfer is in progress.

If the application program calls an E²PROM function while another E²PROM function is still in progress, parameters may be overwritten and an erroneous result will be obtained.

There are 2 exceptions on this rule. When the functions "E2PROM_rd_byte_pol" or "E2PROM_wr_byte_pol" are called, parameters are passed to different registers in the registerbank, the E2PROM status is stored and the transfer is done by polling.

Note that the E2PROM_BUSY bit is not the same as EWP-flag in the ECNTRL register. For write operations the EWP-flag indicates when the writing/erasing of a byte to the E²PROM is finished. The E2PROM_BUSY flag indicates when a complete block of data (e.g. from the E2PROM_wr_block function) has been written to the E²PROM.

Calling sequence:

```
bit Status;  
Status = E2PROM_Status;
```

Function prototype:

```
bit E2PROM_Status (void)
```

Parameters:

None

Driver for 8xC851 E2PROM

MOR952 EIE/AN91009

2.5 E2PROM_wr_byte

Function description:

This function will write a byte to E²PROM. If the source byte has the same value as the byte in the E²PROM, no write action will take place.

Byte transfer is done on interrupt basis. The status of the transfer can be checked with the "E2PROM_Status" function.

This function will automatically enable the E²PROM interrupt. The application program should take care of the E²PROM/UART interrupt enable (with E2PROM_int_en) and the EA bit.

Calling sequence:

```
E2PROM_wr_byte (Src_Byte, Dest_Ptr);
```

Function prototype:

```
void E2PROM_wr_byte (data char Src_Byte, data char Dest_Ptr)
```

Parameters:

Src_Byte:	Byte to be written to E ² PROM
Dest_Ptr:	Address of E ² PROM

Driver for 8xC851 E2PROM

MO935128EIE/AN91009

2.6 E2PROM_rd_byte

Function description:

This function will read a byte from E²PROM. The status of the transfer can be checked with the "E2PROM_Status" function.

Calling sequence:

data char Result;
Result = E2PROM_rd_byte (Src_Ptr);

Function prototype:

char E2PROM_rd_byte (data char Src_Ptr)

Parameters:

Src_Ptr: Address of E²PROM

Driver for 8xC851 E2PROM

EIE/AN91009

2.7 E2PROM_wr_block

Function description:

This function will write a block of data from internal RAM to E²PROM.

Byte transfer is done on interrupt basis. The status of the transfer can be checked with the "E2PROM_Status" function.

This function will automatically enable the E²PROM interrupt. The application program should take care of the E²PROM/UART interrupt enable (with E2PROM_int_en) and the EA bit.

If the source bytes are the same as the bytes in the E²PROM, no write action will take place.

This function will automatically generate ROW-erases, whenever this will reduce programming time.

If during execution of this function, the destination address to the E²PROM is equal to the beginning

of an E²PROM row (3 least significant addressbits are '0') and at least 8 more bytes have to be

programmed, a check will be done whether a ROW-erase will reduce programming time.

If no ROW-erase is done, the time to program the ROW will be:

$$T_{\text{prog}} = A \cdot t_w + B \cdot (t_E + t_w) \quad \text{A: Byte in E}^2\text{PROM is 0x00; source byte in RAM is not 0x00}$$

B: Byte in E²PROM is not 0x00; source byte in RAM <> E²PROM byte

If a ROW-erase is done, programming the ROW will take:

$$T_{\text{prog}} = t_E + C \cdot t_w$$

C: Source byte in RAM <> '0'

Because the erase time (t_E) and the write time (t_W) are equal, the function will do a ROW-erase if

$$A+2.B-C-1 \geq 0$$

Calling sequence:

E2PROM_wr_block (Src_Ptr, Dest_Ptr, Nr_Bytes);

Function prototype:

```
void E2PROM_wr_block (data char *data Src_Ptr, data char Dest_Ptr, data char Nr_Bytes)
```

Parameters:

Src_Ptr:	Address pointer to internal RAM
Dest_Ptr:	Address of first E ² PROM byte
Nr_Bytes:	Number of bytes to write to E ² PROM

Driver for 8xC851 E2PROM

EIE/AN91009

2.8 E2PROM_rd_block

Function description:

This function will read a block of data from E²PROM and store it in internal RAM. The status of the transfer can be checked with the "E2PROM_Status" function.

Calling sequence:

E2PROM_rd_block (Src_Ptr, Dest_Ptr, Nr_Bytes);

Function prototype:

void E2PROM_rd_block (data char Src_Ptr, data char *data Dest_Ptr, data char Nr_Bytes)

Parameters:

- Src_Ptr: Address of first E²PROM byte
- Dest_Ptr: Address pointer to internal RAM
- Nr_Bytes: Number of bytes to read from E²PROM

Driver for 8xC851 E2PROM

EIE/AN91009

2.9 E2PROM_wr_byte_pol

Function description:

This function will write a byte from internal RAM to E²PROM.

If the source byte has the same value as the byte in the E²PROM, no write action will take place.

If an E²PROM function is in progress (except E2PROM_rd_byte_pol), this function will be interrupted but its status will be saved so that the interrupted function will be resumed when the E2PROM_wr_byte_pol function is finished.

Byte transfer is done by polling.

This function may be used e.g. in interrupt service routines, where the possibility exists that the interrupted main program has already started an E²PROM transfer.

Calling sequence:

E2PROM_wr_byte_pol (Src_Byte, Dest_Ptr);

Function prototype:

void E2PROM_wr_byte_pol (data char Src_Byte, data char Dest_Ptr)

Parameters:

Src_Byte:	Byte to be written to E ² PROM
Dest_Ptr:	Address of E ² PROM

Driver for 8xC851 E2PROM

EIE/AN91009

2.10 E2PROM_rd_byte_pol

Function description:

This function will read a byte from E²PROM.

If an E²PROM function is in progress (except E2PROM_wr_byte_pol), this function will be interrupted but its status will be saved so that the interrupted function will be resumed when the E2PROM_rd_byte_pol function is finished.

This function may be used e.g. in interrupt service routines, where the possibility exists that the interrupted main program has already started an E²PROM transfer.

Calling sequence:

data char Result;
Result = E2PROM_rd_byte_pol (Src_Ptr);

Function prototype:

char E2PROM_Rd_Byte_Pol (data char Src_Ptr)

Parameters:

Src_Ptr: Address of E²PROM

Driver for 8xC851 E2PROM

EIE/AN91009

2.11 E2PROM_block_erase

Function description:

This function will erase all 256 E²PROM bytes.

The erase function is done on interrupt basis. The status of the transfer can be checked with the "E2PROM_Status" function.

This function will automatically enable the E²PROM interrupt. The application program should take care of the E²PROM/UART interrupt enable (with E2PROM_int_en) and the EA bit.

Calling sequence:

E2PROM_block_erase();

Function prototype:

void E2PROM_block_erase (void)

Parameters:

None

Driver for 8xC851 E2PROM

EIE/AN91009

2.12 E2PROM_security_on

Function description:

This function inhibits access to E²PROM from external program memory.

The following scheme gives the access possibilities when this function is executed.

EA pin	Address of E2PROM access instruction	Access to E2PROM
1	< 4096	YES
1	>= 4096	NO
0	< 4096	NO
0	>= 4096	NO

The write function is done on interrupt basis. The status of the transfer can be checked with the "E2PROM Status" function.

This function will automatically enable the E²PROM interrupt. The application program should take care of the E²PROM/UART interrupt enable (with E2PROM_int_en) and the EA bit.

Calling sequence:

```
E2PROM_security_on();
```

Function prototype:

```
void E2PROM_security_on (void)
```

Parameters:

None

Driver for 8xC851 E2PROM

E2PROM EIE/AN91009

2.13 E2PROM_security_off

Function description:

This function will remove E²PROM protection. Access to E²PROM from external program memory is now possible if this function is executed from the right program memory.

The following scheme gives the possibilities when 'E²PROM_security_off' is executed after completion of the 'E²PROM_security_on' function.

The following table assumes that the address at which 'E²PROM_security_off' resides is smaller than 4096.

EA pin	Address of E2PROM access instruction	Mode	Access to E2PROM	E2PROM erased
1	< 4096	0	YES	NO
1	>= 4096	0	YES	NO
1	< 4096	1	YES	YES
1	>= 4096	1	YES	YES
0	< 4096	0	NO	NO
0	>= 4096	0	NO	NO
0	< 4096	1	YES	YES
0	>= 4096	1	YES	YES

The following table assumes that the address at which 'E²PROM_security_off' resides is greater than 4096.

EA pin	Address of E2PROM access instruction	Mode	Access to E2PROM	E2PROM erased
1	< 4096	0	YES	NO
1	>= 4096	0	NO	NO
1	< 4096	1	YES	YES
1	>= 4096	1	NO	YES
0	< 4096	0	NO	NO
0	>= 4096	0	NO	NO
0	< 4096	1	YES	YES
0	>= 4096	1	YES	YES

Driver for 8xC851 E2PROM

8xC851 E2PROM

Calling sequence:
E2PROM_Security_Off;

Function prototype:
void E2PROM_Security_Off (data char Mode)

Parameters:
Mode: If '0', then the protection will only be removed when this function is executed from internal program memory.
When executed from external memory the protection remains.
If '1', then the protection can also be removed when this function is executed from external memory, however all E2PROM bytes will be erased.

EA pin	Address of E2PROM access instruction	Mode	Access to E2PROM	E2PROM erased
1	< 4096	0	YES	NO
1	> 4096	0	YES	NO
1	< 4096	1	YES	YES
1	> 4096	1	YES	YES
0	< 4096	0	NO	NO
0	> 4096	0	NO	NO
0	< 4096	1	YES	YES
0	> 4096	1	YES	YES

EA pin	Address of E2PROM access instruction	Mode	Access to E2PROM	E2PROM erased
1	< 4096	0	YES	NO
1	> 4096	0	NO	NO
1	< 4096	1	YES	YES
1	> 4096	1	NO	YES
0	< 4096	0	NO	NO
0	> 4096	0	NO	NO
0	< 4096	1	YES	YES
0	> 4096	1	YES	YES

Driver for 8xC851 E2PROM

EIE/AN91009

3: PROGRAM EXAMPLES

3 examples are given that show how to use these functions with C, PL/M51 and assembly programs. In the examples, a string of characters is written to and read from E²PROM. When reading back the string, spaces are replaced by underscores.

3.1 C example

The disk contains the file E2PROM.H that should be included in the C application program. E2PROM.H contains the function prototypes of the E²PROM functions. The example program can be found on the disk in file TEST_C.C

When the application module is compiled and assembled, it should be linked to the E²PROM function module E2PROM.OBJ and the UART interrupt handler UART.OBJ.

3.2 PL/M51 example

The disk contains the file E2PROM.DCL that should be included in the PL/M51 application program. E2PROM.DCL contains the external function declarations for the E²PROM functions. The example program can be found on the disk in file TEST_PLM.PL

When the application module is compiled and assembled, it should be linked to the E²PROM function module E2PROM.OBJ and the UART interrupt handler UART.OBJ. When linking, the linking control 'NOCASE' must be used!

3.3 Assembly example

The disk contains the file E2PROM.MAC which contains the macro definitions of the functions. Including these macro's in the source file, eases programming.

E.g. the sequence

```
MOV _E2PROM_rd_block_BYTE ,Src_Ptr      ;Pointer to E2PROM
MOV _E2PROM_rd_block_BYTE+1, Dest_Ptr   ;Pointer to RAM
MOV _E2PROM_rd_block_BYTE+2, #Nr_Bytes  ;Number of bytes to transfer
LCALL _E2PROM_rd_block                  ;Call function
```

can be replaced by

```
%E2PROM_rd_block(Src_Ptr, Dest_Ptr, #Nr_Bytes)
```

The file E2PROM.GLO contains the EXTRN-definitions of the functions and constants that are used by the driver. Only the definitions used by the source program should be included.

When the application module is compiled and assembled, it should be linked to the E²PROM function module E2PROM.OBJ and the UART interrupt handler UART.OBJ.

Driver for 8xC851 E2PROM

EIE/AN91009

3.4 Listing of examples

LISTING C EXAMPLE:

```

#include "E2PROM.h"
#include <string.h>
#define E2PROM_Base_Address 0x58

rom char Txt_tab[] = "This is an E2PROM test for 8xC851";

void main(void)
{
    data char    Data_Buffer[35];
    data char    Count;

    E2PROM_init();          /* Initialise E2PROM */
    E2PROM_int_en(0x01);    /* E2PROM interrupt level 1 */
    EA=1;                  /* Global interrupt enable */

    romidmove(&Data_Buffer,&Txt_tab,sizeof(Txt_tab)-1); /* Copy string from ROM
                                                         to RAM */
    E2PROM_wr_block(&Data_Buffer,E2PROM_Base_Address,sizeof(Txt_tab)-1);
                                                         /* Copy string to E2PROM */

    /* Time to do other usefull things while data is being written to
       E2PROM on interrupt basis */

    while (E2PROM_status()); /* Wait till transfer to E2PROM is finished */

    /* Read string from E2PROM and replace spaces " " by underscores " _ " */
    for (Count=0;Count != sizeof(Txt_tab)-1;Count++)
    {
        /* Read E2PROM byte */
        Data_Buffer[Count] = E2PROM_rd_byte(E2PROM_Base_Address+Count);
        if (Data_Buffer[Count] == ' ')
            Data_Buffer[Count] = '_';
    }

    E2PROM_block_erase(); /* Erase E2PROM */
    /* Time to do other things while erasing */

    while (E2PROM_status()); /* Wait till erasing is finished */
    EA=0;
}

```

Driver for 8xC851 E2PROM

EIE/AN91009

LISTING PL/M51 EXAMPLE

```

$DEBUG
$CODE

E2PROM: Do;
$INCLUDE (E2PROM.DCL)
$INCLUDE (UTIL51.DCL)

Test: Do;

    Declare E2PROM_Base_Address literally '58H';
    Declare Txt_tab(*) Byte Constant
        ('This is an E2PROM test for 8xC851');
    Declare Data_Buffer(35) Byte Main;
    Declare Count Byte Main;

    Call E2PROM_init;          /* Initialise E2PROM */
    Call E2PROM_int_en(01);    /* E2PROM interrupt level 1 */
    Enable;                    /* Global interrupt enable */

    /* Copy string from ROM to RAM */
    Call MOVCD1(.Txt_tab,.Data_Buffer,length(Txt_tab));

    /* Copy string to E2PROM */
    Call E2PROM_wr_block(.Data_Buffer,E2PROM_Base_Address,length(Txt_tab));

    /* Time to do other usefull things while data is being written to
       E2PROM on interrupt basis */

    Do While E2PROM_status = 1; /* Wait till transfer to E2PROM is finished */
    End;

    /* Read string from E2PROM and replace spaces " " by underscores "_" */
    Do Count=0 To length(Txt_tab);
        /* Read E2PROM byte */
        Data_Buffer(Count) = E2PROM_rd_byte(E2PROM_Base_Address+Count);
        If (Data_Buffer(Count) = ' ') Then Data_Buffer(Count) = '_';
    End;

    Call E2PROM_block_erase; /* Erase E2PROM */
    /* Time to do other things while erasing */

    Do While E2PROM_status = 1; /* Wait till erasing is finished */
    End;
    Disable;

    End Test;
End E2PROM;

```

Driver for 8xC851 E2PROM

EIE/AN91009

LISTING ASSEMBLY EXAMPLE

```

$DEBUG
$CASE

;=====
;
; INCLUDE FILE : E2PROM.GLO
; PACKAGE      : E2PROM
;=====

EXTRN CODE    (_E2PROM_init)
EXTRN CODE    (_E2PROM_int_en)
EXTRN NUMBER  (_E2PROM_int_en_BYTE)

EXTRN CODE    (_E2PROM_status)
EXTRN CODE    (_E2PROM_rd_byte)
EXTRN NUMBER  (_E2PROM_rd_byte_BYTE)

EXTRN CODE    (_E2PROM_wr_block)
EXTRN NUMBER  (_E2PROM_wr_block_BYTE)

EXTRN CODE    (_E2PROM_block_erase)

;=====
; Include macro definitions
;=====
$INCLUDE (E2PROM.MAC)

        BUFFER SEGMENT DATA
        RSEG BUFFER
Data_Buffer:    ds 35
Count:          ds 1
Stack:          ds 15

        TABLE SEGMENT CODE
        RSEG TABLE
Txt_tab:        db    "This is an E2PROM test for 8xC851"

E2PROM_Base_Address    EQU 58H
Length_Txt              EQU 33

        CSEG AT 00          ;Reset vector
        LJMP MAIN

        TEST_ASM SEGMENT CODE
        RSEG TEST_ASM

```

Driver for 8xC851 E2PROM

EIE/AN91009

```

MAIN:    MOV SP,#Stack-1      ;Initialize stack pointer
         %E2PROM_init         ;Initialize E2PROM
         %E2PROM_int_en(#01)  ;E2PROM interrupt level 1
         SETB EA              ;Enable global interrupt

         MOV DPTR,#Txt_tab     ;Initialise pointers to copy Txt_tab to RAM
         MOV R0,#Data_Buffer
         MOV R2,#Length_Txt

COPY_LOOP:
         CLR A                ;Copy Txt_tab to RAM
         MOVC A,@A+DPTR        ;Get byte from ROM
         MOV @R0,A             ;Store in RAM
         INC DPTR              ;Update pointers
         INC R0
         DJNZ R2,COPY_LOOP     ;Check if all copied

         ;Write data to E2PROM
         %E2PROM_wr_block(#Data_Buffer,#E2PROM_Base_Address,#Length_Txt)
         ;
         ; Time to do other usefull things while data is being written to
         ; E2PROM on interrupt basis
         ;

NEW_CHECK:
         %E2PROM_status        ;Wait till transfer to E2PROM is finished
         JC NEW_CHECK

         ;Read string from E2PROM and replace spaces " " by underscores "_"
         MOV R0,#Data_Buffer   ;Initialise pointers
         MOV R1,#E2PROM_Base_Address
         MOV R2,#Length_Txt

READ_LOOP:
         %E2PROM_rd_byte(R1)
         MOV @R0,A             ;Store byte in RAM
         CJNE A,#" ",NEXT_READ ;Check if byte is " "
         MOV @R0,#"_"          ;If yes, replace with "_"

NEXT_READ:
         INC R0                ;Update pointers
         INC R1
         DJNZ R2,READ_LOOP

         %E2PROM_block_erase   ;Erase E2PROM
         ;Time to do other things while erasing */

NXT_CHECK:
         %E2PROM_status        ;Wait till transfer to E2PROM is finished
         JC NXT_CHECK
         CLR EA                ;Disable interrupts
         JMP $                 ;End of program

END

```

Driver for 8xC851 E2PROM

EIE/AN91009

4: DEBUG MACROS

The disk contains some debug macro's that ease the debugging of programs that use the 8xC851 E²PROM. These macro's can be executed by the XRAY51 High Level Language debugger. The user can read from and write to E²PROM bytes without programming the individual sfr's.

Before the macro's can be executed, they must be loaded by XRAY51. This will be done automatically if the file 'E2PROM.INC' is included when invoking XRAY51 or during a debug session. E2PROM.INC will load the macros and define some symbols used by the macros. If not all macro's are used, the file E2PROM.INC can be edited to prevent the loading of these macro's. This may be necessary when there is insufficient memory to load the macro's, because e.g. other macro's have been loaded. Another advantage of only loading the relevant macro's is reduction of loading time.

When a macro is called from the debugger, the following sfr's will remain unchanged: ECNTRL, EADRL, EADRL and ETIM. During macro execution, all interrupts will be disabled. Access to the E²PROM with the macros is independent of the state of the security bit. The execution and results of the macro are visible on the I/O screen of XRAY51 (VSCREEN 3).

Read(Start address, Stop address):

The value of E²PROM bytes from 'START ADDRESS' to 'STOP ADDRESS' will be shown. If 'START ADDRESS' <= 'STOP ADDRESS' only the value of 'START ADDRESS' will be shown.

Write(Start address, Stop address, Value):

The E²PROM bytes from 'START ADDRESS' to 'STOP ADDRESS' will be programmed with 'VALUE'. If 'START ADDRESS' > 'STOP ADDRESS', no E2PROM bytes will be programmed. If the ETIM register contains the value 0x08, it is considered that ETIM is not initialized. The macro will give a warning, and return to the debug screen.

Copyto(Ram address, E2PROM address, Count):

Macro will copy 'COUNT' bytes, starting from internal RAM address 'RAM ADDRESS' to the E²PROM, starting at address 'E2PROM ADDRESS'.

If during copying, the RAM address becomes > 0x7F or the E²PROM address becomes > 0xFF, copying will be stopped and a warning is given that an address limit is reached.

If the ETIM register contains the value 0x08, it is considered that ETIM is not initialized. The macro will give a warning, and return to the debug screen.

Copyfrom(E2PROM address, Ram address, Count):

Macro will copy 'COUNT' bytes from E²PROM address 'E2PROM ADDRESS' to the internal RAM, starting at address 'RAM ADDRESS'.

If during copying the RAM address becomes > 0x7F or the E²PROM address becomes > 0xFF, copying will be stopped and a warning is given that an address limit is reached.

Erase():

All E²PROM bytes will be erased.

If the ETIM register contains the value 0x08, it is considered that ETIM is not initialized. The macro will give a warning, and return to the debug screen.

Driver for 8xC851 E2PROM

EIE/AN91009

5 CONTENTS OF DISK

The disk contains the following 3 directories:

- 1: **USER**
This directory contains the files that may be included or linked to the source program.
 - E2PROM.ASM :Source file of E²PROM driver
 - E2PROM.OBJ :Object file of E²PROM driver
 - E2PROM.H :Header file for C applications
 - E2PROM.DCL :Declaration file for PL/M51
 - E2PROM.MAC :Macro definitions for assembly applications
 - E2PROM.GLO :Global definitions for assembly applications
 - UART.SRC :UART interrupt handler (will only clear flags; user should customize it)
 - UART.OBJ :Object file of UART interrupt handler
- 2: **DEBUG**
This directory contains the macros and include file used with XRAY51 debugger.
 - E2PROM.INC :Include file that reads macro files in XRAY51
 - *.MAC :XRAY51 macro's
- 3: **EXAMPLE**
This directory contains the source files of the example programs described in the note.
 - TEST_C.C :C example
 - TEST_PLM.PL :PL/M51 example
 - TEST_ASM.ASM :ASM51 example

Low RF-emission applications with a P83CE654 microcontroller

EIE/AN92001

1. Introduction

Quite recently customers asked for dedicated 8-bit microcontrollers at high clockrates which would not cause Radio Frequency (RF-) disturbances in consumer and communication applications, e.g. key-board control, tuning, etc...

Up to now either a slower 4-bit microcontroller was used or a more modern 8-bit microcontroller is in use with additional RF-shielding and filtering measures.

With a carradio manufacturer, a bargain was set for a new (EMC friendly) 8-bit microcontroller such that the product would maintain equal receiving performance with more control features. The new receiver would then only be modified for the microcontroller part.

To set the emission requirements, for a microcontroller in this application, the measurement technique described in the application note, EIE/AN91001 "Workbench EMC evaluation method", was used and applied to the existing receiver in which the new microcontroller has to fit in.

Later on, the same technique was used to verify the basic and modified samples which contained one or more measures to reduce RF-emission.

Each (mask) shrinking event will cause circuits to become faster and produce more RF-disturbances for these kind of appliances. As such, more EMC measures need to be taken after each shrinking event to maintain the above set emission requirements.

2. Measures to reduce RF-emission

Up to now, evident measures are known to reduce RF-emission. Within a few years time, new techniques will mature to keep pace with shrinking actions.

So far it is only interesting to take emission reducing measures on-chip for a microcontroller when there are no external ROM, RAM or (E)EPROM-busses. The externally required data- and address-busses will cause much more RF-radiation, when used in a non-shielded way.

We've restricted ourselves to stand-alone controllers, which can control most functions locally with, when required low speed serial busses e.g. I²C (100 kbit/s or 400 kbit/s with output-edge-control), or parallel communication to another controller.

To compare our results, reference is made to an existing standard microcontroller based on the INTEL 80C51-core mounted in a 40 pin DIL package.

Dedicated software has been used to allow true comparison between all microcontrollers in a defined application.

Low RF-emission applications with a P83CE654 microcontroller

EIE/AN92001

2.1 IC-package

Density problems demand smaller packages to allow further integration of functions within a certain product size. It was decided to take the Quad Flat Pack (QFP) 44, because it was the lowest pin number package available above the 40 pins required to apply the circuit as stated above.

The advantage of this package is the smaller loop area enclosed by the currents running through the circuit. As a result, direct radiation will be reduced. The worst-case area difference between the package sizes is:

DIL 40:	48,46 x 15,24 x 4,3 mm (l x w x h, h = PCB + die-path height)
diagonal:	50,8 mm
looparea:	218,5 mm ²
QFP 44:	11,4 x 8,8 x 1,2 mm (l x w x h, h = die-path height)
diagonal:	14,4 mm
looparea:	17,3 mm ²

When package radiation is considered only, the magnetic dipole moment produced by the package will be reduced by a factor of 12,6, which can give a decrease in electromagnetic emission of about 22 dB.

External supply decoupling capacitors can be placed more closely to the pins where needed. The latter will shorten the length of the current path between Vdd and Vss, thus resulting in a lower voltage drop appearing in-between reference point taken on the PCB. Considering this, the current path reduction will be a factor of 4,5 which would result in an RF-emission reduction of about 12 dB. One should consider that also I/O-pins will contribute to the RF-emission.

Practical radiation figures have shown a decrease of about 12 dB.

2.2 IC-pinning

As already given above, supply and I/O pinning will determine emission performance, due to the fact that radiation is determined by the way currents flow through a device. For some years it is known that each output pin should be embedded in-between a Vdd and a Vss pin. As such, the 3 one-byte wide busses would need $3 \times (2 \times 8 + 1)$ pins (O, Vdd, Vss), equals at least 51 pins. For such a device this seems impractical.

The advantage of such a pinning will be that currents will always flow through adjacent leadframe fingers and as such, emission will be absolutely minimal. For this application, I/O will commonly occur at low frequencies, and as such, their contribution to RF-emission will be low. In CMOS applications these currents will mainly occur during transitions by charging and discharging the output load.

A more serious contribution will arise from the ground-bounce or supply-bounce which will occur between the PCB-reference and the IC's-substrate. This disturbance voltage will be superimposed to all I/O's which are either coupled to Vss or Vdd. When these lines are long, longer than the path involved for supply decoupling, their contribution to radiation

Low RF-emission applications with a P83CE654 microcontroller

EIE/AN92001

can be high. The disturbance source, being the supply- or ground-bounce voltage, has a negligible impedance (ωL , L = leadfinger + bondingwire inductance) and will be difficult to filter by using simple and cheap components such as capacitors. More often, radiation will increase by such measures due to the increased current through these I/O's.

The most effective action will be the use of several ground and supply pins. The ground pins must be spread around the circumference of the package while the supply pins need to be the adjacent to these grounds to benefit the mutual coupling in-between. This mutual coupling reduces the 'effective' series inductance with the external decoupling capacitor.

With the DIL 40, pin 20 is the Vss-pin as pin 40 is the supply pin, Vdd. The I/O-pins are randomly located. With the QFP-44, for the Vss the following pins are selected: 6, 16, 28, 39 and the supply pins are: 17 (for I/O) and 38 (for the core).

Another cause for ground-bounce can be the X-tal oscillator's output with its external capacitance. Normally, this contribution can be reduced by adding some series impedance with the output and changing the capacitors values such that the X-tal circuit operation remains within its linear range with sufficient amplitude.

All together, the ground-bounce, which will be emitted by the I/O, can be reduced by a factor of 4 (4 gnd pins QFP versus 1 gnd pin DIL), assuming random I/O current distributions. Individual decoupling of the I/O and core supply will dampen the supply bounce even further.

As such these measures will reduce emissions further by some 12 dB. The expectation of even more drastic effects can be accounted for by the shorter leadframe finger lengths comparing DIL 40 to QFP 44.

2.3 On-chip decoupling measures

From the above, it will be clear that RF-emission will primarily come from the core and that the lower frequencies will be mainly caused by the I/O. For the latter output-edge-rate control can be considered, when the number of outputs are high compared to the number of Vss and Vdd pins.

When core decoupling is integrated, the high frequency low energy currents can close their loop on silicon. As a result these RF-currents will not flow through the leadframe any longer and will not add to any additional ground-bounce.

If the latter is implemented without further considerations, the effect can be quite negative. The original circuit design assumed that all charge (current) comes from the external decoupling capacitor. This charge (current) then, flows from the capacitor, through its leadwires, PCB traces, IC leadframe, interconnect to the circuit (that needed it) and then back. When on-chip decoupling is used, charge is there and switching will occur instantaneously.

In our case, most bus-driving circuits were re-designed such that waveforms maintain within their specifications under worst-case conditions (temperature, supply voltage, etc.).

The overall result is that on-chip decoupling will require little space due to the fact that all

Low RF-emission applications with a P83CE654 microcontroller

EIE/AN92001

empty areas can be used, even within the circuit-blocks, and that bus-driving circuits can be made much smaller with respect to the present dimensions.

2.4 PCB design measures

The QFP-package and the pinning of the controller will only reduce the RF-emission when the PCB is laid out following some constraints.

An example of a good supply and ground plane lay-out is given in Fig.1. The microcontroller is mounted on the component side of a double layer PCB. The supply pairs Vss1/Vdd1 for the I/O-supply and Vss3/Vdd2 for the core supply are connected directly to the ceramic chip capacitors C1 and C2, which are surface mounted on the solder side. A short connection of Vss3/Vdd2 to the capacitor C2 minimizes loop inductance. Thus, the external supply current drawn by the core will flow in this loop mainly. This current path can be ensured by the insertion of an inductor (L2) in series to the +5V general supply. An equal action is taken for the I/O-supply.

The implemented on-chip decoupling allows a separation between core and I/O-ports. The PCB lay-out shall use these Vdd/Vss connections to minimize the loop areas in-between signal lines from each port pin to any load via the ground plane back to these Vss/Vdd-pins.

By applying these hints, Vss2 may be used as the return pin for ALE, PSEN, port0 and port2 because Vss2 is connected very near to them. Vss3 shall be used for the core supply mainly. Vss1 is nearest to lower part of port2, upper part of port3 and the crystal oscillator. Even though Vss1 may be the best return for port2 and port3, this pin shall in any case be used as return for the external crystal oscillator capacitors (not shown here). Vss4 is located nearest to the lower part of port3 and the whole port1, being the best return for these ports.

3 Future developments

Existing products are shrunk, to cut costs and increase complexity. Recent developments (SAC3 → SAC2 → SAC1, C300 → C250 → C200 and others) have demonstrated an upwards tendency in the RF-emission following shrinking when EMC is not considered. This means that the end in taking measures to reduce RF-emission has not been reached. Further improvements are still possible already considered for new products, such as:

- a PLL-circuit, to replace the high frequency X-tal oscillator,
- output-edge-control (application dependent),
- further circuit improvements on-chip e.g. coplanar supply, decoupling measures within cell-blocks, multi-phase clock systems to prevent simultaneous switching

4 Results

All measurements were carried out under the same conditions, using the same kind of PCB, with the same software and the same bus loadings, Fig.2.

Low RF-emission applications with a P83CE654 microcontroller

EIE/AN92001

Tested were the following applications:

- | | | | |
|-----|------------------|--|-------------|
| ☛ 1 | 87C51, DIL 40, | 12 MHz X-tal | → 0 dB(rel) |
| ☛ 2 | 80C31, QFP 44, | but with 2 Vss and only one Vdd connection, 12 MHz X-tal | → 13 dB |
| ☛ 3 | 83CE654, QFP 44, | with Address Latch Enable (ALE) active, 12 MHz X-tal | → 32 dB |
| | | 4 Vss and 2 Vdd pins as given above. | |
| ☛ 4 | 83CE654, QFP 44, | ALE off, 12 MHz X-tal | → 50 dB |
| ☛ 5 | 83CE654, QFP 44, | ALE off, externally 12 MHz, 500 mV sinewave. | → 54 dB |

5 Conclusions

With the measures indicated in chapter 2, RF-emission can be reduced substantially, especially for stand-alone microcontroller applications. Up to now, a number of measures have been implemented which indicate a improvement of about 40 dB in the FM-region when changing from a DIL 40, annex 1 to a QFP 44 application, annex 4, taking into account minor additional supply measures. The inclusion of a PLL can make the improvement even further to about 50 dB, annex 5.

With these measures, some 12 to 22 dB can be accounted for by the choice of the package, same 12 dB due to the pinning and the rest due to the internal measures.

The advantage of these measures can be easily wasted by insufficient measures on the PCB. Constraints are given in chapter 2.4.

When considering the reducing effects PCB-layouts might have to RF-emission, the following relative information needs to be considered:

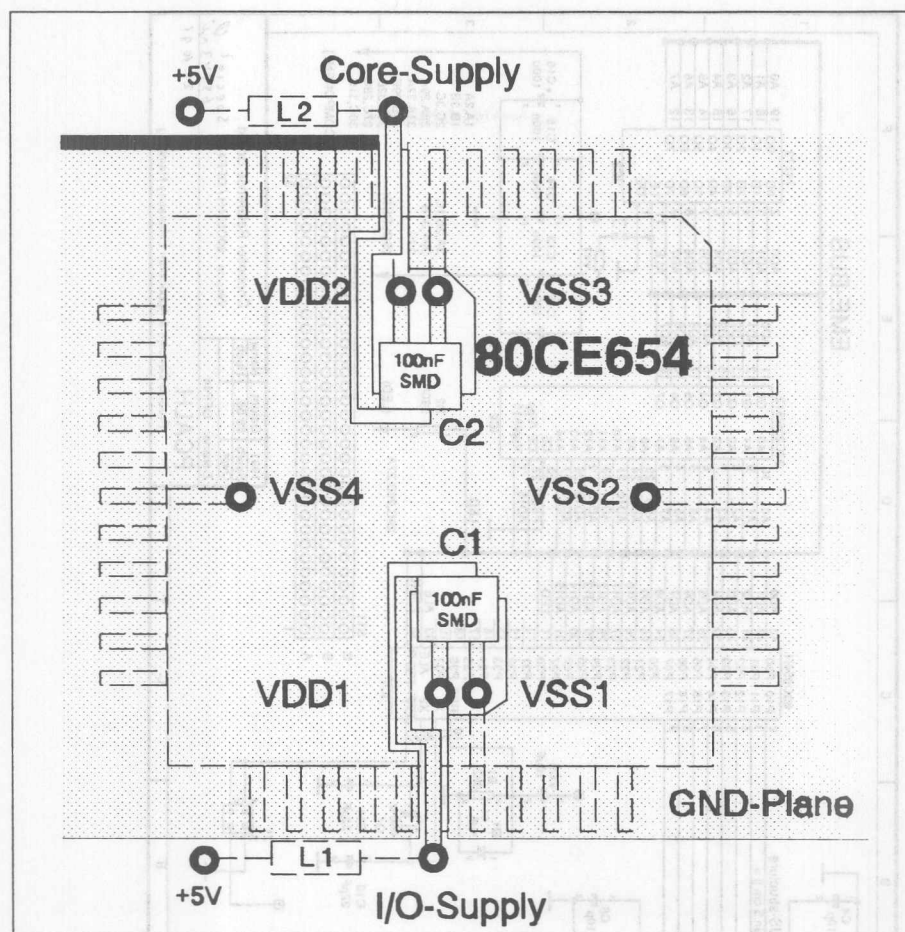
- | | | |
|---|--------------------|------------------------------|
| ☛ | single layer board | → 0 dB (relative) |
| ☛ | double layer board | → 26 dB (best case) |
| ☛ | multi layer board | → 44 dB (4-layer, best case) |

6 References

- [1] Syllabus of the design course "Fast digital and analog circuit design", Philips CTT, 1991, Eindhoven
- [2] IC package outlines, Philips Components, 1990, 12NC: 9398 175 80011.
- [3] Improvements in microcontrollers for a better EMC behaviour, H.Schutte, EIE/IN90039 version 2, 1991
- [4] Investigations on EME improvements for the microcontrollers 8xCE592 and 8xCE598, H-W. Lütjens, HKI/IR 92001, 1992

Low RF-emission applications with a
P83CE654 microcontroller

EIE/AN92001



Low RF-emission applications with a P83CE654 microcontroller

EI/AN92001

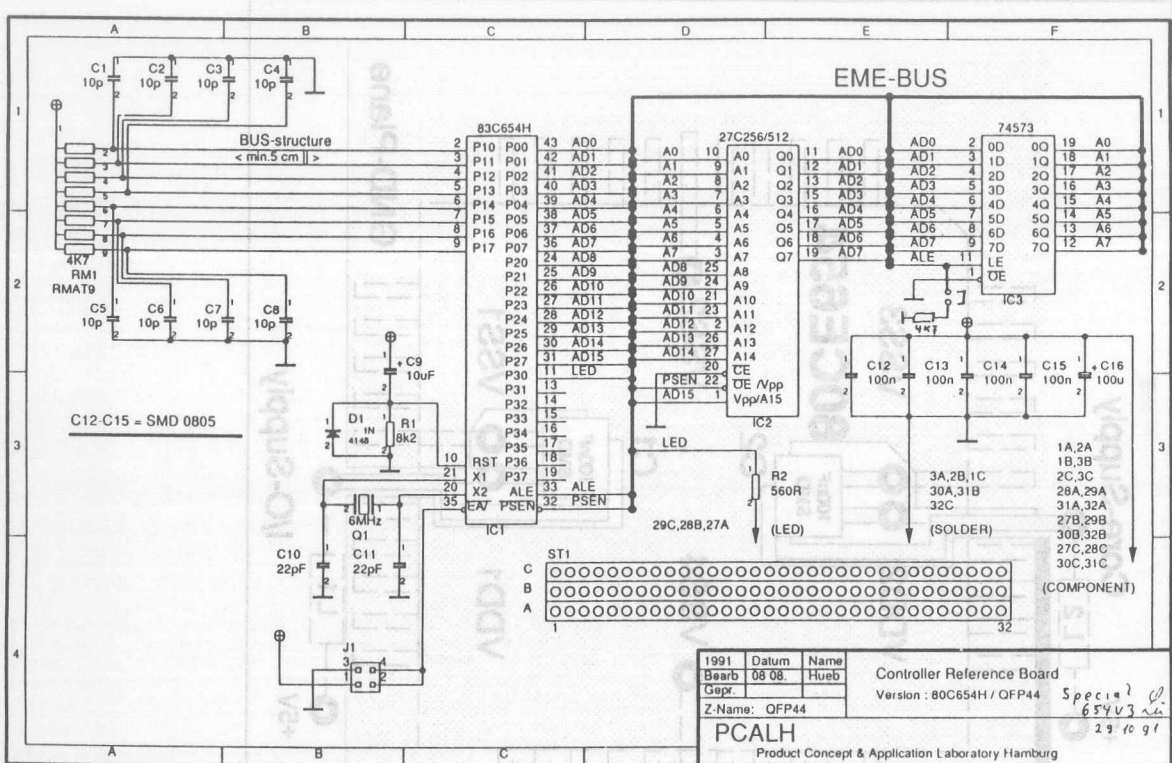
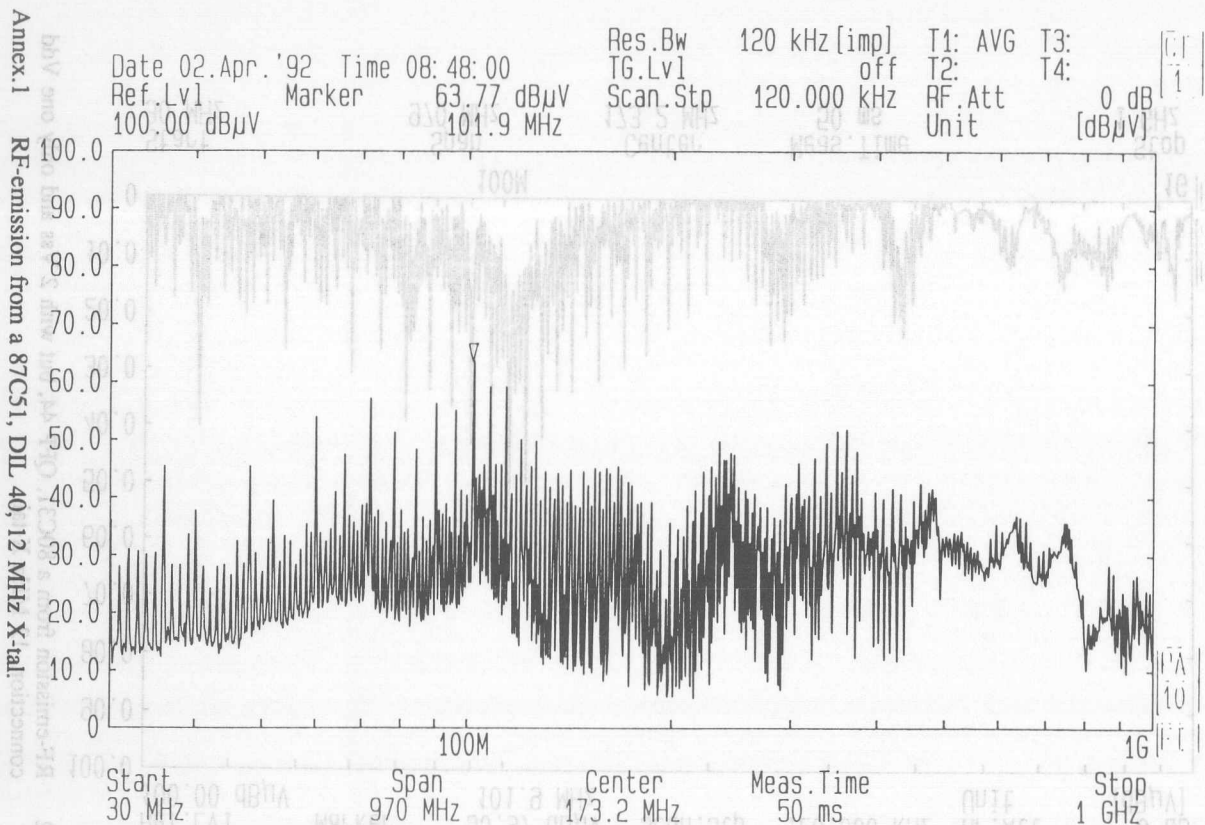


Fig.2 Circuit diagram of tested applications.

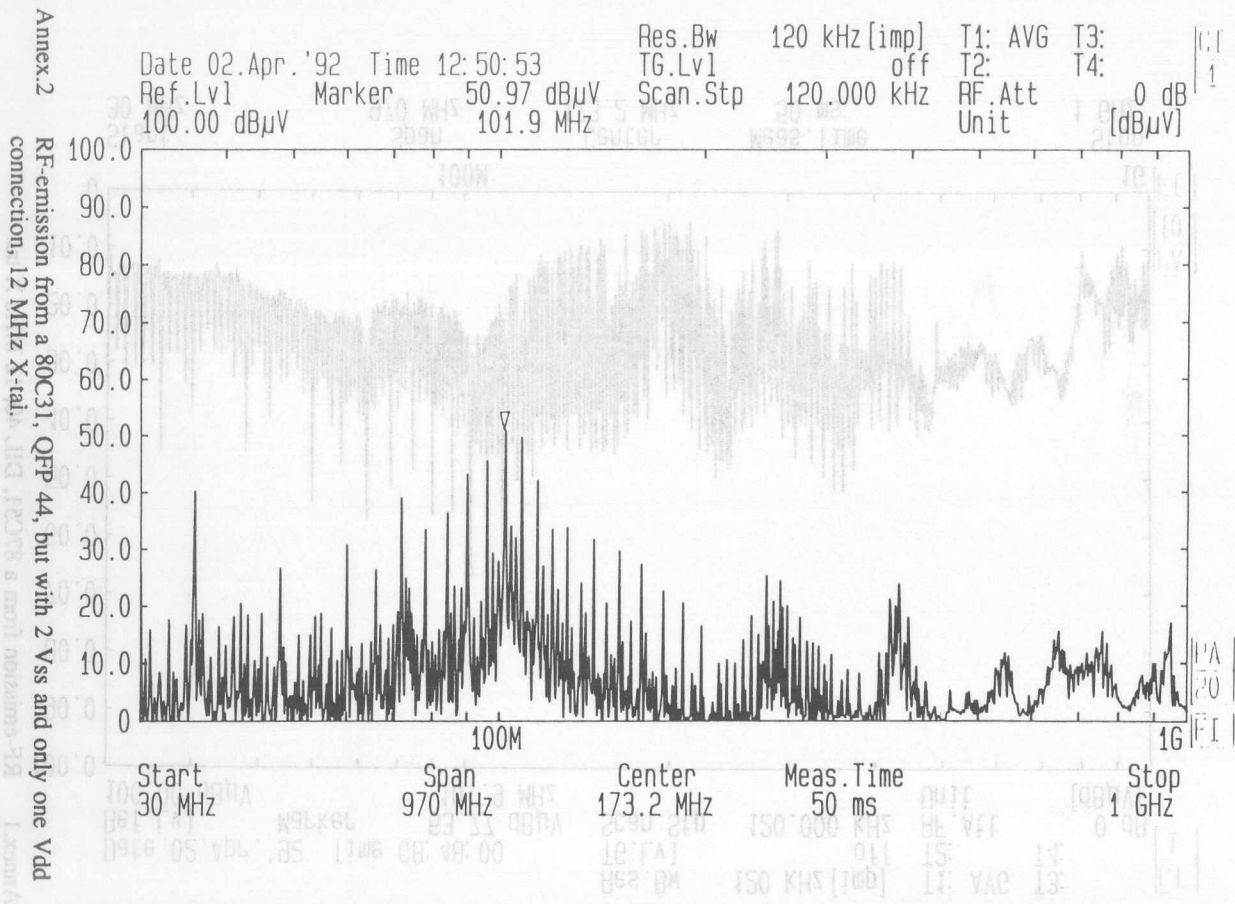
Low RF-emission applications with a P83CE654 microcontroller

EIE/AN92001



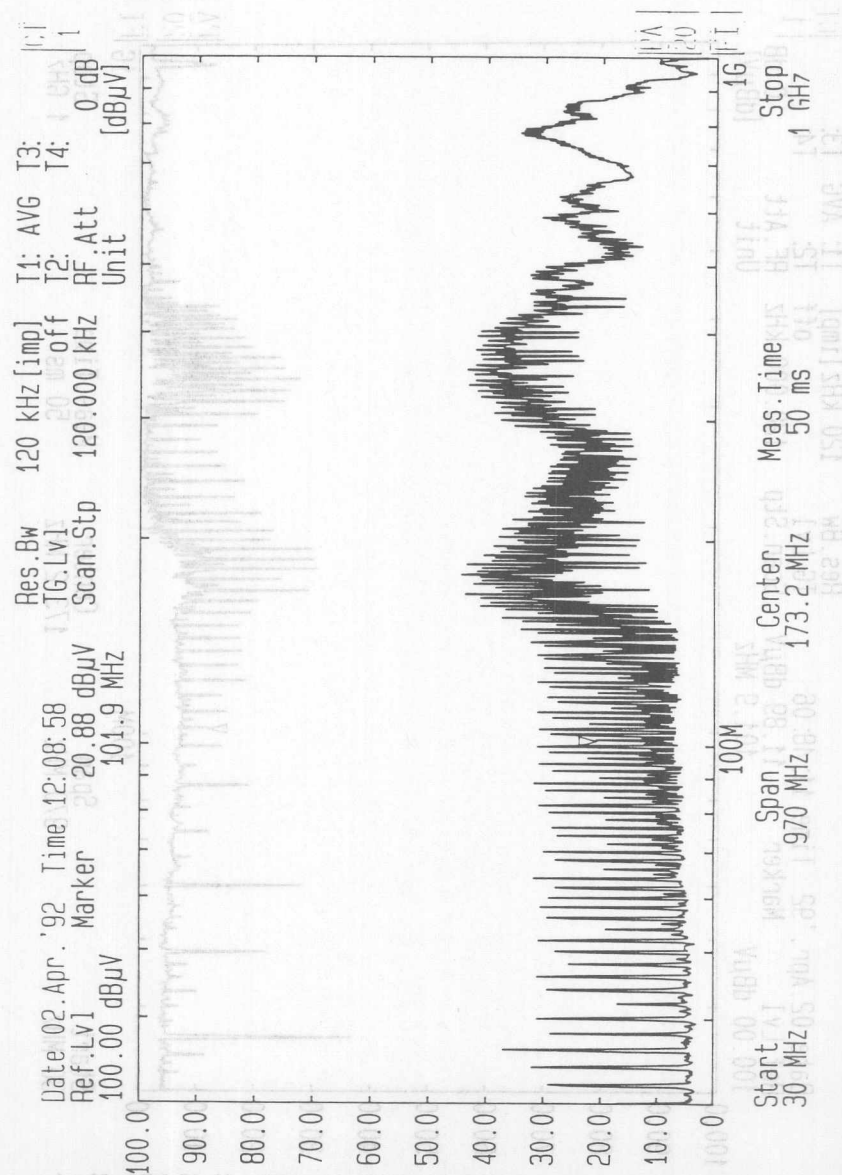
Low RF-emission applications with a P83CE654 microcontroller

EIE/AN92001



Low RF-emission applications with a P83CE654 microcontroller

EIE/AN92001

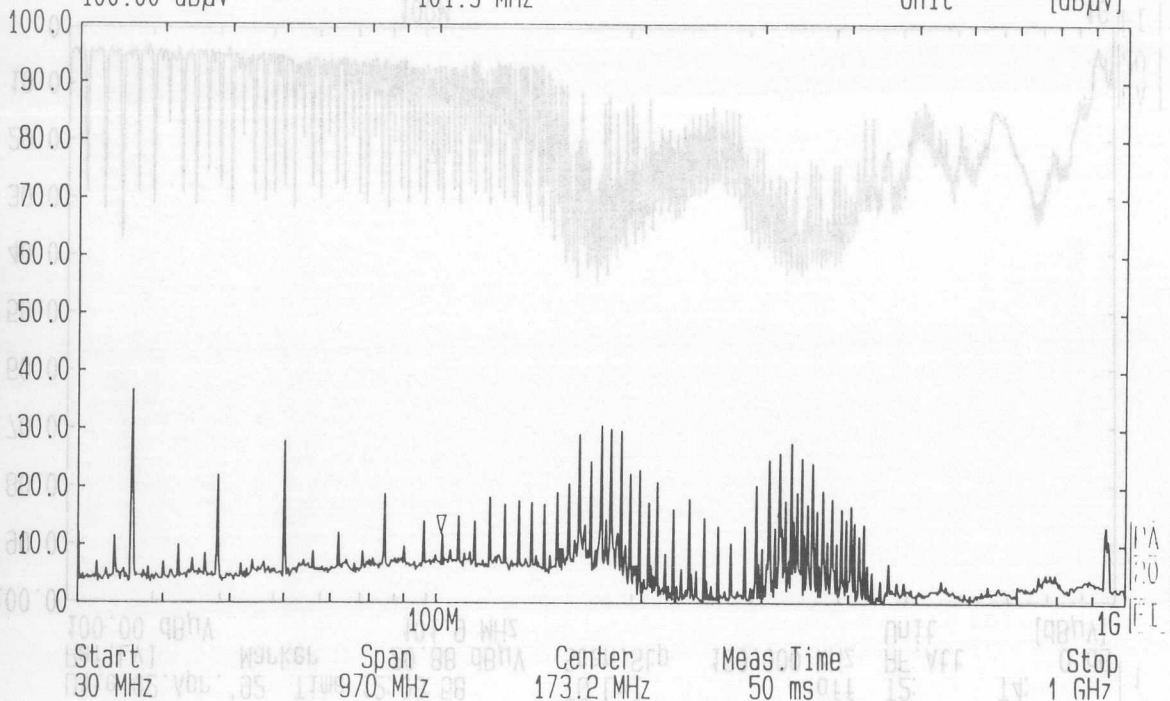


Annex.3 RF-emission from a 83CE654, QFP 44, with Address Latch Enable (ALE) active, 12 MHz X-tal. 4 Vss and 2 Vdd pins as given above.

Low RF-emission applications with a P83CE654 microcontroller

EIE/AN92001

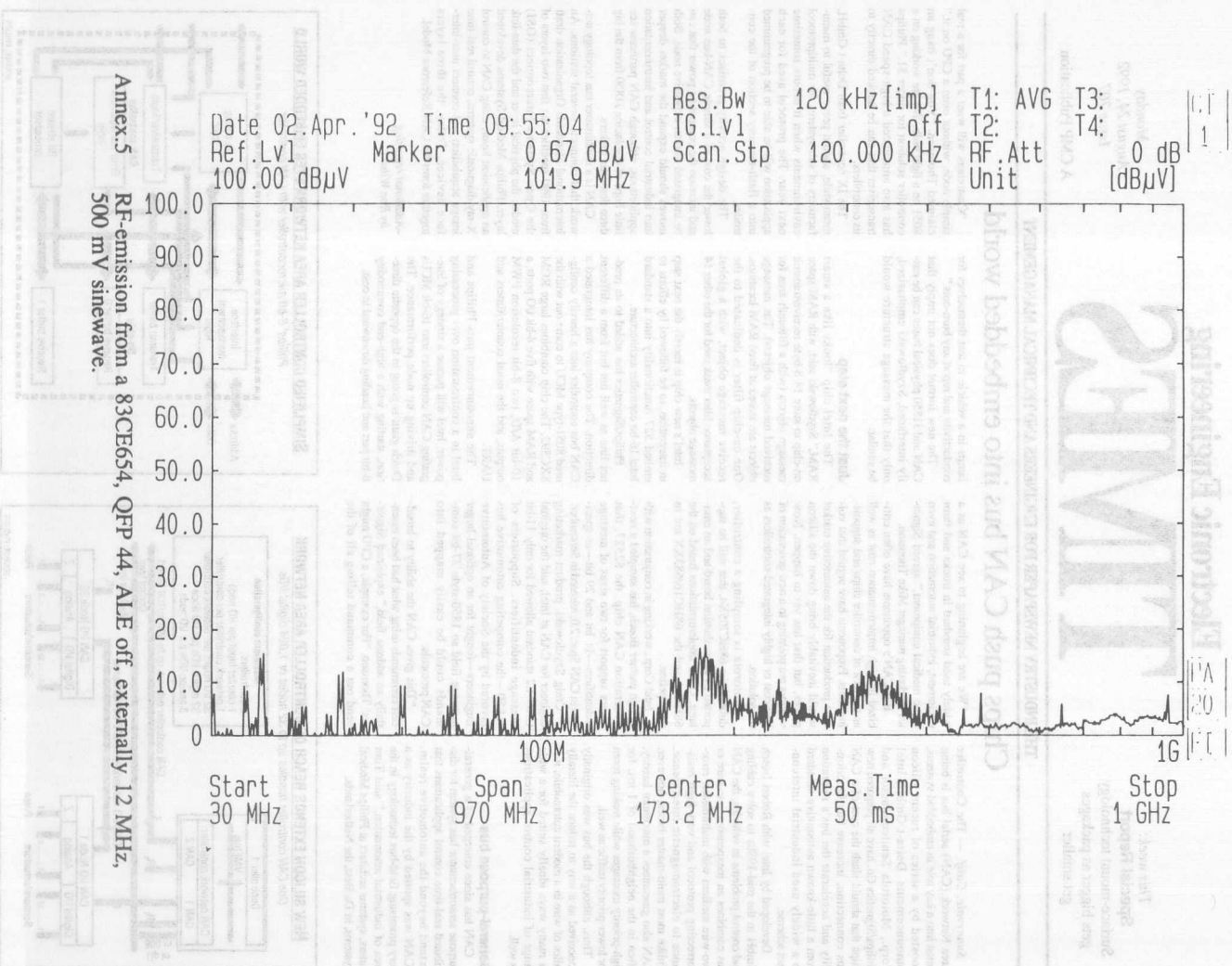
Date 02.Apr.'92 Time 11:48:06 Res.Bw 120 kHz [imp] T1: AVG T3:
Ref.Lvl 100.00 dBμV Marker 11.89 dBμV T6.Lvl off T2: T4:
Scan.Stp 120.000 kHz RF.Att Unit 0 dB [dBμV]



Annex 4

RF-emission from a 83CE654, QFP 44, ALE off, 12 MHz X-tal.

Low RF-emission applications with a P83CE654 microcontroller



Chips push CAN bus into embedded world

This week:
Special Report
 Surface-mount technology
 gets bigger as packages
 get smaller

Electronic Engineering TIMES

Monday
 August 24, 1992
 Issue 707

A CMP Publication

THE INDUSTRY NEWSPAPER FOR ENGINEERS AND TECHNICAL MANAGEMENT

Chips push CAN bus into embedded world

Sunnyvale, Calif. — The Controller Area Network (CAN) serial bus is being thrust into a key role in embedded systems, boosted by a series of recent silicon announcements. Delta-t GmbH, Intel Corp., Motorola Semiconductor and Philips/Signetics Co. have all prepared new chips that should slash the cost of a CAN bus connection, increase bus functionality and accelerate the bus's migration from a little-known automotive standard to a widely used industrial interconnect scheme.

Developed by Intel with Robert Bosch GmbH in the mid-1980s to solve cabling and control problems in vehicles, the CAN bus combines an inexpensive, one-wire or two-wire medium with multimaster, error-correcting protocol and very high resistance to electromagnetic interference. Unlike most multi-master buses, however, CAN also guarantees a maximum latency, often in the neighborhood of 1 ms, for high-priority messages while making room for lower-priority traffic as well.

Thus, although the bus was originally conceived as a way to reduce the literally miles of wire in a modern automobile, it is in many ways ideally suited for a wide range of industrial control applications as well.

General-purpose bus

CAN has since emerged as a general-purpose sensor/actuator bus system for distributed real-time control applications that extend beyond the automotive realm. "CAN was spotted by the industry as a very promising field-bus technology in the area of industrial automation," said Tom Suters, systems architect at Philips Medical Systems, in Da Best, the Netherlands.

"We are beginning to see CAN as a widely used standard in trucks and farm equipment, industrial automation and even some medical equipment," agreed Signetics marketing manager Mike Thomson.

But CAN's opponents have often criticized its high implementation cost as well as the lack of controller chips and supporting tools. Proponents have argued that volume production for the automotive market would inevitably bring down the silicon prices, but that has yet to happen. Now vendors are pointing to a new generation of low-cost or highly integrated controllers as the solution.

Motorola is sampling a controller, dubbed the 68HC705X4, that will be supported by an evaluation board and an emulator. Other implementations based on the 68HC11 and the 68HC16/683XX are in the pipeline.

Intel Corp., working in conjunction with design partner Bosch, has sampled a next-generation CAN chip, the 82527, that offers support for two sizes of message identifiers—11 bit and 29 bit—as specified in CAN Spec. 2.0, released in September.

Craig Szydlowski, product marketing engineer for CAN at Intel, said the original CAN 1.2 standard allowed for only 11-bit message identifiers. Supporters of J1850—the competing automotive bus favored by the Society of Automotive Engineers—fought for an optional larger message field so 1850-style 27-bit commands could be easily mapped into CAN protocols.

"This gives CAN the ability to broadcast commands using what had been essentially an address field," explained Signetics' Thomson. "For example, a CPU might send out a command telling all of the

lamps in a vehicle to test themselves for conductivity and report any burn-outs."

The new format does not imply that CAN and J1850 physical buses can be easily interfaced, Szydlowski emphasized, only that the message structure would be similar.

Just the next step

The Intel chip "looks like a smart RAM," Szydlowski said, with RAM space on-chip to store 14 8-byte receive/transmit message objects (with a fifteenth area for received message objects). The message objects are stored at fixed RAM locations. One on-chip filter is dedicated to the receive message object, with a global acceptance filter mask used for the other 14 message objects.

Intel's new chip is merely the next step in integration, to be followed by efforts to embed 527 functionality into a standard Intel 16-bit controller architecture.

Philips/Signetics has added to its product line as well but from a different direction. The company has integrated a CAN bus controller into a heavily configured 8051-type MCU to come out with the 8XC592. The chip combines large ROM and RAM space with five 8-bit I/O ports, a 10-bit A/D, two 8-bit-resolution PWM outputs, and the usual counter/timers and UART.

The announcement puts Philips and Intel in a confrontation over processing power. Intel will pursue a strategy of integrating CAN controllers into 16-bit MCUs and driving up node performance. The Dutch giant is going in the opposite direction, starting with a high-end commodity 8-bit part and heading downward in cost.

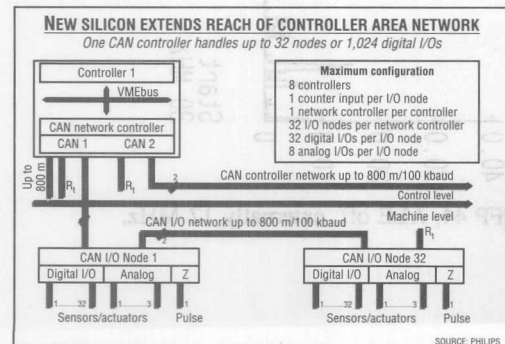
"Customers will want a part for a real simple node, without even a CPU on it," claimed Thomson. "You don't hang an 8051 on a light bulb. We are working on a controller solution for under \$1." Philips has also announced a high-speed CAN transceiver that can be hooked directly to its controllers.

The IX controller from Delta-t GmbH, meanwhile, should prove useful to manufacturers of sub-systems for multiprotocol environments when it debuts sometime next year. The protocol used for each application will be able to be programmed into a flash-memory section of the controller.

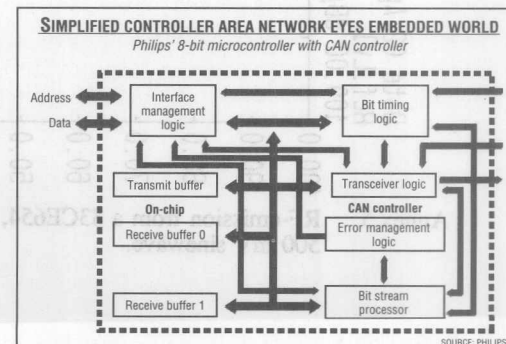
The design activity promises to both lower the cost of a simple CAN-bus node and increase the computing power that can be integrated into an expensive node. Both moves should expand the market deeper into industrial control and instrumentation applications, although CAN partisans see little hope of displacing J1850 from the big three domestic makers.

CAN is a multimaster bus topology network that connects several stations. An International Standards Organization draft from 1990 specifies the first two layers of the Open Systems Interconnect (OSI) model: the physical layer and the data-link layer. Philips Medical Systems developed an application layer, using CAN to control X-ray diagnostic systems, control real-time image acquisition and connect user-interface devices. Today, the three layers together form the CAN Reference Model.

—Additional reporting
 by Ron Wilson



Handling CAN-controller complexity has been a barrier to acceptance for this control-oriented bus.



Chips push CAN bus into embedded world

NEWS

Controller Area Network (CAN) Products From Philips Semiconductors

8XC592 CAN Microcontroller - The 8XC592 is a stand-alone high-performance microcontroller based on the 80C51 architecture. Its on-chip CAN interface makes it ideal for applications with a harsh, noisy environment.

The **8XC592 Features** include:

- 16K x 8 EPROM (87C592)
- 16K x 8 ROM (83C592)
- ROMless (80C592)
- 512 x 8 RAM, expandable externally to 64k bytes
- Three Standard 16-bit timer/counters
- A 10-bit ADC with 8 Multiplexed Analog Inputs
- Two 8-bit Resolution, Pulse Width Modulation Outputs
- Five 8-bit I/O Ports Plus One 8-bit Input Port Shared with Analog Inputs
- CAN Controller with DMA Transfer between Internal Data RAM and CAN Registers
- Standard 80C51 UART
- On-Chip Watchdog Timer

82C200 CAN Interface - The 82C200 is a highly integrated stand-alone controller for CAN. The 82C200 with a simple bus line connection performs all the functions of the physical and data-link layers. The application layer of an electronic control unit (ECU) is provided by a microcontroller, to which the 82C200 provides versatile interface.

The **82C200 Features** include:

- Multi-Master Architecture
- Interfaces with a Large Variety of Microcontrollers
- 2032 Message Identifiers
- Powerful Error Handling Capability
- Configurable Bus Interface

82C150 Serial Linked I/O CAN Interface - The 82C150 Serial Linked I/O CAN is a single-chip 16-bit I/O device with an on-chip CAN-controller. 82C150 is a very cost-effective way of increasing the I/O-capability of a microcontroller as well as reducing the amount and complexity of wiring. Advanced safety provided by the CAN protocol combined with low-cost makes the 82C150 a very attractive product for a wide variety of applications.

The **82C150 Features** include:

- Single-Chip I/O Device with CAN Protocol Controller
- 16 Configurable Digital or Analog I/O Ports
- Each Port Individually Configurable via the CAN bus
- 10-bit A/D converter with up to 6 Multiplexed Inputs
- Bit Rate 20 kbit/sec to 125 kbit/sec



Add Text Overlay to Any Video Display

CIRCUIT CELLAR **I N K**®

THE COMPUTER APPLICATIONS JOURNAL

October/November, 1992 — Issue #29

MEASUREMENT & CONTROL

SPECIAL SECTION:
*Embedded
Graphics & Video*

Time Domain Reflectometer
Overlay Text on Video
X-10 Interfacing



\$3.95 U.S.
\$4.95 Canada

Add Text Overlay to Any Video Display

Add Text Overlay to Any Video Display

SPECIAL SECTION

Bill Houghton

S

Suppose, for the moment, you've built and installed the HCS II home control system described primarily in issues 25 and 26 (February/March and April/May '92) of the *Computer Applications Journal*. In issue 27 (June/July '92), Ed Nisley described an add-on LCD output device as a way to obtain information about the status of the system and its various nodes. It's a nice addition to the network, but useful only when you're near to the display module. What do you do if you're across the room watching TV settled into your favorite armchair? You could get up and venture across the room. Or, if you build the interface described in this article, you could hit a button on your HCS II IR remote and

see network information displayed on your TV set overlaid onto the program you are watching.

This article describes an On-Screen Display (OSD) terminal for HCS II (we'll call it "TV-Link" to match the other HCS II modules) that allows color text characters to be displayed on top of a background color video signal. The terminal is built around the Philips/Signetics 87C054 OSD microcontroller.

FEATURES OF 87C054

The 87C054 is an 80C51-based microcontroller designed to provide an advanced OSD for TV and video applications. It can produce characters in eight foreground and eight background colors. In addition, the background color can be removed, showing through the original video. It also has nine pulse-width modulator outputs for controlling analog functions. Similar to a standard 80C51, it has 28 digital I/O pins, two external interrupts, and two timer/counters. RAM and ROM spaces on the 87C054 are larger than the 80C51: 192 bytes of RAM and 16K bytes of EPROM. (The OSD has *additional* RAM and EPROM areas that are not part of the normal 80C51 memory map.)

One unique feature of the 87C054 is what Signetics describes as a "soft-

All the latest VCRs have on-screen programming in an effort to simplify the notorious task. Now you can have your HCS II or any other computer send you messages while you're watching your favorite TV show.

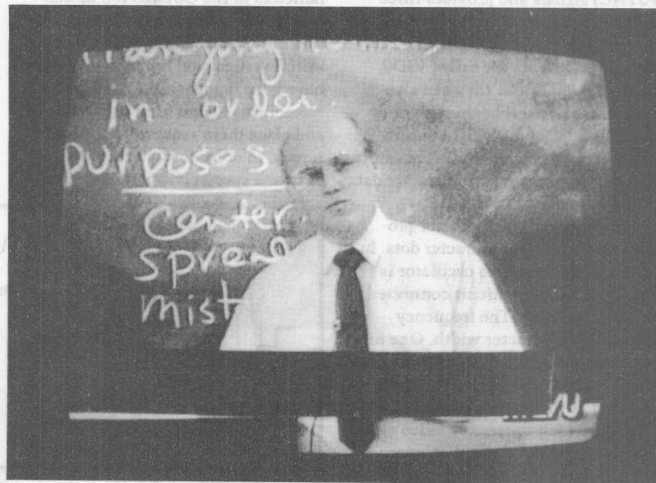


Photo 1—The TV-Link can be used by your computer to overlay messages on any video signal.

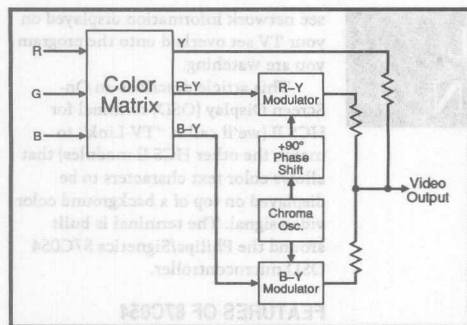


Figure 1—Converting from RGB to NTSC involves modulating and adding difference signals.

ware ADC." This ADC consists of an internal 4-bit DAC that feeds one input of a comparator. The other comparator input can be connected to one of four I/O pins. The output of the comparator is tied to a status bit in a register that is testable by software. A TV set often uses this logic for measuring the AGC voltage during tuning. A real-time clock and other low-precision analog measurements can also use it as a zero-crossing detector.

The OSD of the 87C054 consists of a 128-character RAM array (OSD-RAM), a 64-character font EPROM, a video clock oscillator, and the OSD logic. The OSD logic accepts horizontal sync (*HSYNC*) and vertical sync (*VSYNC*) signals and provides three digital video outputs for character information. In the datasheet for this part, these outputs are called *VID0*, *VID1*, and *VID2*, but they can also (and perhaps better) be thought of as *RED*, *GREEN*, and *BLUE*. A multiplexer control output is also present to indicate when to display character data or original video information.

The video clock oscillator provides timing for the character dots. In most applications, this oscillator is simply an LC tank circuit connected to the *VCLK* pins. The frequency controls the character width. One nice feature is that the video clock is killed at the leading edge of *HSYNC* and restarted at the trailing edge of *HSYNC*, which causes the video clock to start in the same phase on every line, ensuring the dots align vertically from one scan line to the next.

The character font stores the binary pattern for the individual characters. Characters are 14 dots wide and 18 scan lines high.

The OSDRAM stores the characters to be displayed on the screen along with certain attribute data pertaining to those characters. Once a character has been written

to the OSD, no further CPU intervention is required to "refresh" the screen.

Many OSD architectures have been developed over the years for use in the consumer television market. Almost all of them have required fixed character row formats, limiting the designer's flexibility in designing video menus and screens.

The 87C054 was designed to avoid such constraints, and there are no architectural limits on the number of characters in a row of text or the number of rows of text to a screen. (There are physical limits imposed by the dot clock frequency and the scan rate, of course.)

The *HSTART* and *VSTART* parameters in the *OSORG* (on-screen origin) register define the initial position of the start of the OSD. Once the initial vertical and horizontal positions have been found, the 87C054 will "fetch" characters from the OSDRAM and place them sequentially on the screen. In order to have multiple rows of text, a special character has been

defined and is referred to as *NEWLINE*. The *NEWLINE* character is much like a carriage return/line feed sequence on a computer in that it terminates the current row of characters and starts a new row of text. One advantage of this architecture is that it eliminates the need to pad display memory with space characters. The fetching and painting of rows of text will continue until either a new vertical sync pulse is detected or until an *END* attribute is fetched along with a *NEWLINE* character.

NOW FOR THE DETAILS...

Now that you understand the concepts of an OSD operation and the capabilities of the 87C054, focus your attention on the details required to overlay characters onto live video.

The 87C054 OSD has a multiplexer output for switching video sources. Simply switching between the input video signal and the OSD character data would be nice. Unfortunately, you can't because the input video (from our home entertainment center) is in NTSC format and the character data is in RGB format. (Keep in mind that the goal is to input live video, add on-screen text, and present the result as a video signal at the output of our circuit.)

One solution is to decode the input video into separate red, green, blue, *HSYNC*, and *VSYNC* signals. Then you could perform the multiplexing between video information and character data in RGB format. The resultant signals could then be encoded back into baseband video. If there were other reasons for the conversion into RGB, such a conversion

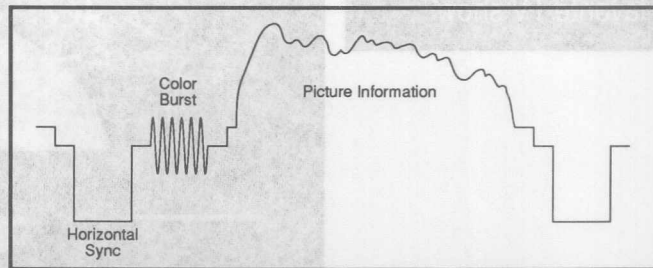


Figure 2—A typical line of NTSC video consists of an initial horizontal sync pulse, followed by a short color burst signal, then the actual picture information.

Add Text Overlay to Any Video Display

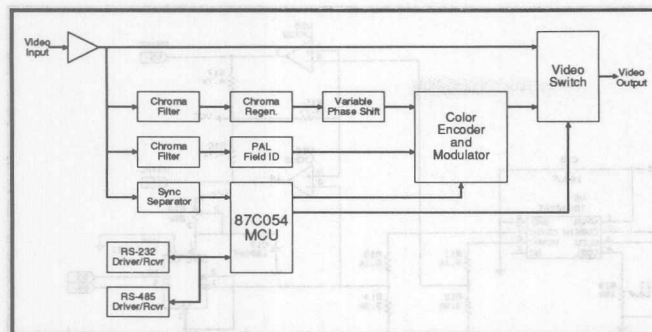


Figure 3—At the core of the TV-Link is the Signetics 87C054 microcontroller. The unit supports both RS-232 and RS-485 communications with a host computer.

would be the way to go. However, the process of converting to RGB and then converting back to video introduces some distortion that could be visible on the screen.

Another solution is to find a way to encode the RGB data from the OSD microcontroller into video. Then you can simply switch between the two sources. Sound simple? The situation gets a little more complex when you consider the issues of making the characters appear with the proper color in NTSC. Reviewing how color is

encoded in NTSC is in order.

COLOR TELEVISION

When you first look at video, you often wonder why in the world it was done the way it was. A long time ago, before Americans had ever heard the names of Japanese TV makers, RCA research labs were developing color television. One of the requirements imposed on designers by the FCC was that the broadcast signal needed to be compatible with existing black-and-white TV sets and had to be contained

within the same bandwidth as a B/W signal. Such requirements meant that some component of the signal had to contain overall brightness information, which is the main reason why they could not simply transmit separate R, G, and B channels within the video bandwidth allowed. To make a very long story short, the engineers involved decided that the scene brightness (which they called "Y" or luminance) could be described by the relationship

$$Y = 0.59G + 0.3R + 0.11B$$

Someone observed that if they took two copies of the luminance signal and subtracted one copy from one of the colors (say, RED) and did the same with a different color (say, BLUE), the result would be two signals that contained all of the information needed to represent color. These resultant signals, R-Y and B-Y, are the color difference signals.

Now that you have two signals, how can they be transmitted on one RF carrier? The answer they came up with was to modulate one of the signals (B-Y) with an RF carrier. The

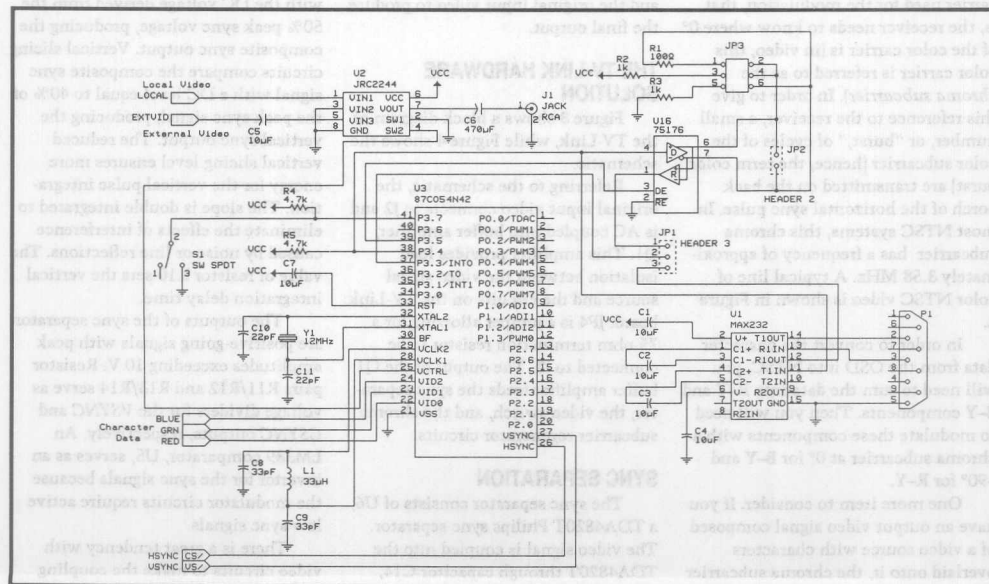


Figure 4a—The JRC2244 switches between the incoming video signal and the on-screen characters under control of the 87C054 MCU.

Add Text Overlay to Any Video Display

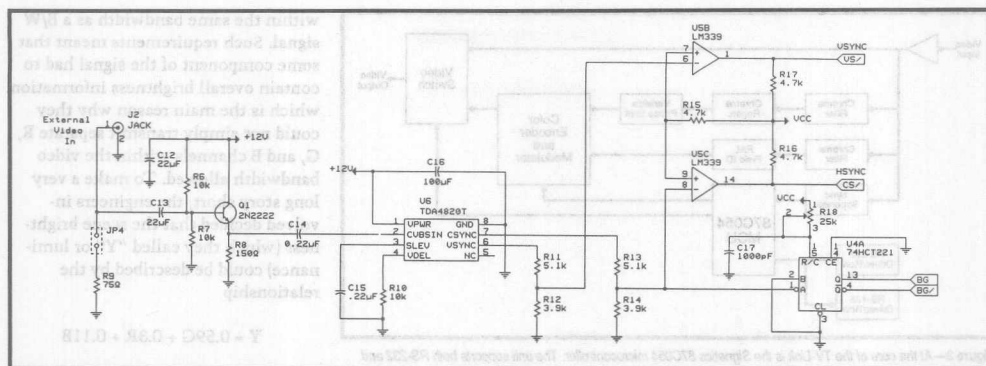


Figure 4b—The TDA4820T sync separator provides the processor with horizontal and vertical sync signals.

other signal (R-Y) was to be modulated with the same RF carrier, but the carrier would be shifted by 90°. When the outputs of the two modulators are added together, the result is the vector sum of the two signals, containing both an amplitude and a direction (phase angle). See Figure 1.

We now have a single signal that contains all of the color information. The TV receiver needs just one more piece of information to demodulate this signal. It needs a reference for the carrier used for the modulation, that is, the receiver needs to know where 0° of the color carrier is (in video, this color carrier is referred to as the *chroma subcarrier*). In order to give this reference to the receiver, a small number, or "burst," of cycles of the color subcarrier (hence, the term color burst) are transmitted on the back porch of the horizontal sync pulse. In most NTSC systems, this chroma subcarrier has a frequency of approximately 3.58 MHz. A typical line of color NTSC video is shown in Figure 2.

In order to convert the character data from the OSD into NTSC, you will need to sum the data into R-Y and B-Y components. Then you will need to modulate these components with a chroma subcarrier at 0° for B-Y and +90° for R-Y.

One more item to consider. If you have an output video signal composed of a video source with characters overlaid onto it, the chroma subcarrier reference (i.e., color burst) present on

the output video signal is the color burst provided in the original input video. In order for the receiver/monitor to interpret the color of the OSD characters correctly, the chroma subcarrier used to modulate the OSD's R-Y and B-Y components must have exactly the same frequency and phase as the color burst on the original video input signal.

Once you get the OSD information in the form just described, you can switch between this "OSD video" and the original input video to produce the final output.

THE TV-LINK HARDWARE SOLUTION

Figure 3 shows a block diagram of the TV-Link, while Figure 4 shows the schematic.

Referring to the schematic, the original input video connects to J2 and is AC coupled into buffer amplifier, Q1. This amplifier provides load isolation between the video signal source and the circuits on the TV-Link board. JP4 is a jumper allowing for a 75-ohm termination resistor to be connected to J2. The output of the Q1 buffer amplifier feeds the sync separator, the video switch, and the chroma subcarrier regenerator circuits.

SYNC SEPARATION

The sync separator consists of U6, a TDA4820T Philips sync separator. The video signal is coupled into the TDA4820T through capacitor C14, where it is amplified with a gain of 15.

The black level clamping voltage is stored in capacitor C14. From the stored black level voltage and the peak sync voltage, the 50% value of the peak sync voltage is generated and stored in capacitor C15. A slicing level control circuit ensures a constant 50% peak sync value regardless of the picture content amplitude provided the sync pulse amplitude is between 50 mV and 500 mV. A comparator in the composite sync slicing stage compares the amplified video signal with the DC voltage derived from the 50% peak sync voltage, producing the composite sync output. Vertical slicing circuits compare the composite sync signal with a DC level equal to 40% of the peak sync signal, producing the vertical sync output. The reduced vertical slicing level ensures more energy for the vertical pulse integration. The slope is double integrated to eliminate the effects of interference caused by noise or line reflections. The value of resistor R10 sets the vertical integration delay time.

The outputs of the sync separator are positive-going signals with peak amplitudes exceeding 10 V. Resistor pairs R11/R12 and R13/R14 serve as voltage dividers for the VSYNC and CSYNC outputs, respectively. An LM339 comparator, U5, serves as an inverter for the sync signals because the modulator circuits require active low sync signals.

There is a great tendency with video circuits to make the coupling capacitors very large to pass the low-

Add Text Overlay to Any Video Display

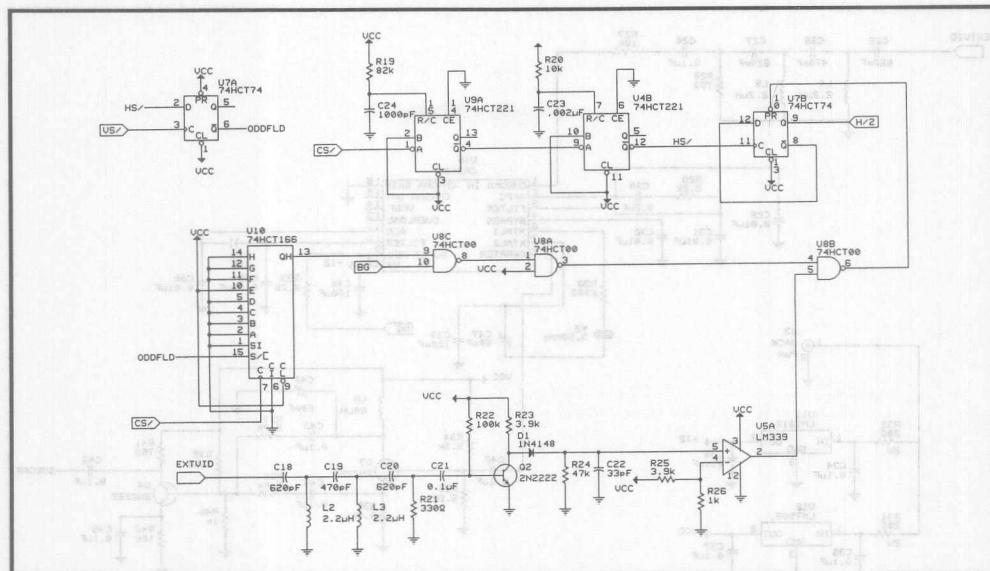


Figure 4c—In order to properly overlay colors onto a PAL signal, you must know whether you're on an odd or an even field, so extra circuitry must be included on the board to support PAL.

frequency sync components (60/50 Hz, typically) into low-impedance nodes. The TDA4820T has a moderately high input impedance on pin 2. Because the black level is stored in C14, the value of C14 should be kept close to 0.22 μ F.

THE 87C054 MCU WITH OSD

The 87C054 microcontroller, U3, accepts composite sync and vertical sync signals from the sync separator and provides RGB digital outputs for character data. The multiplexer control output, VCTRL, connects to the video switch, U2, a JRC2244.

Inductor L1 and capacitors C8 and C9 form a video clock oscillator that determines the width of a character font dot. The values of these components are not critical but are typically chosen such that a video dot width is equal to the spacing between scan lines. This oscillator is killed at the leading edge of the HSYNC signal and allowed to startup at the trailing edge. Such synchronization causes the oscillator to start at exactly the same point from one scan line to the next, causing character dots to appear in exactly the same spot on each line.

In addition to the OSD functions, the 87C054 also performs network interfacing and protocol tasks. This microcontroller has plenty of performance bandwidth because the OSD logic is self-refreshing and independent of the MCU core.

VIDEO SIGNAL SWITCHING

The JRC2244 video switch, U2, contains three video inputs, two of which are used in this application. One of these inputs, VIN1, is capacitively coupled to the OSD video signal. This signal is the 87C054's RGB data after encoding into baseband video. The other input, VIN3, is capacitively coupled to the original video input signal. The JRC2244 provides internal bias sources to provide DC restoration to its video inputs. The JRC2244 accepts a switching control signal from the 87C054 and switches its output between the original video input and the OSD video signal. The video switch also has an internal 75-ohm line driver in its output stage.

The JRC2244 has a moderate input impedance of about 15k ohms, allowing 10- μ F coupling capacitors to

be used. The output coupling capacitor is large because this signal can be used to drive 75-ohm loads.

RGB ENCODING

The LM1886, U13, and the LM1889, U14, encode the RGB data from the 87C054 into baseband video. The LM1886 has three DACs, one for each color. Each of these DACs has 3-bit inputs, but because the 87C054 data is digital, the inputs to the LM1886 DACs are tied together yielding an output for each DAC that is either full-scale or zero. The outputs of the three DACs are internally summed to produce the luminance, R-Y, and B-Y amplitudes. The LM1889 accepts the regenerated chroma subcarrier, modulates the R-Y and B-Y signals, and produces baseband video on pin 13. Transistor Q5 is used as a buffer amplifier with voltage dividers R49 and R50 producing proper levels for the video switch. Note that the LM1889 accepts an external subcarrier signal at the junction of R46 and C52, but this subcarrier undergoes a phase shift caused by the resistor and capacitor networks associated with pins 1

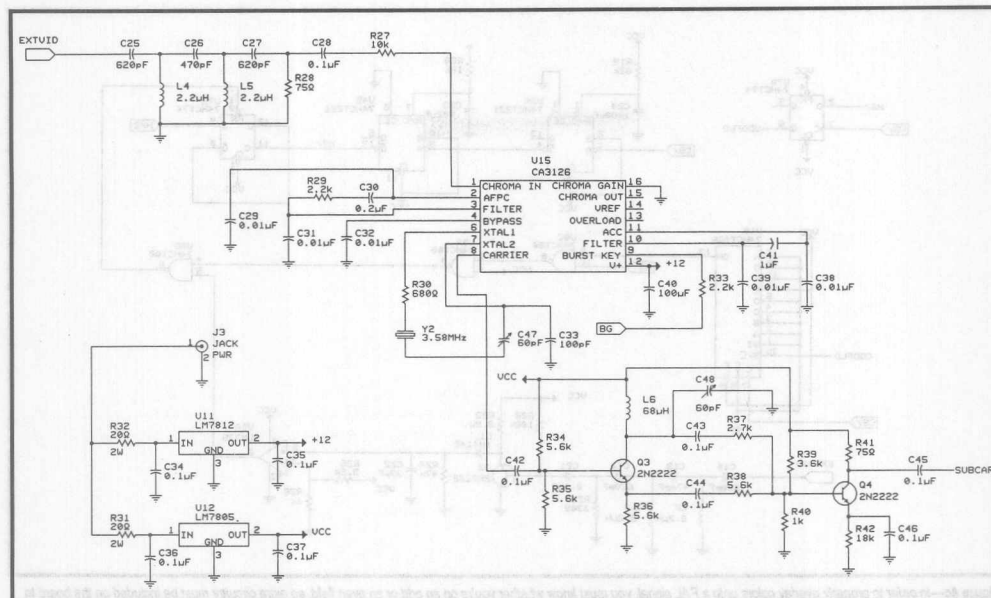


Figure 4d—The CA3126 TV chroma processor is designed specifically for regenerating chroma subcarriers.

and 18 of the LM1889. This phase shift will need to be considered when the subcarrier is regenerated.

CHROMA SUBCARRIER REGENERATION

The circuits that reproduce a chroma subcarrier in the same frequency and phase as the color burst consist of a high-pass filter, a sample-and-hold phase-locked loop (PLL), and a phase shift network and amplifier.

The passive high-pass filter consists of inductors L4 and L5, resistor R28, and capacitors C25, C26, and C27. The filter starts passing signals at about 3.2 MHz, allowing the chroma subcarrier to pass through to the CA3126, U15.

The CA3126 is a TV Chroma Processor IC designed specifically for regenerating chroma subcarriers. This IC contains a VCO and a PLL with sample-and-hold circuits in the error correction loop. As a result, the VCO-generated carrier is compared with the chroma signal from the high-pass filter during the time that color burst is present, indicated by the burst gate pulse (which I will describe later).

The regenerated carrier output is present on pin 8 of the CA3126. Even though this carrier is phase locked to the color burst, it is not at exactly the same phase as the color burst. The nature of a PLL is such that the output will be locked but will always have some constant fixed phase delay relative to the input. Also, recall that the input circuits of the LM1889 added an additional constant phase shift to the injected carrier.

The phase shift network and amplifier consisting of the Q3 and Q4 stages compensate for these fixed phase delays. This circuit provides an output whose phase is adjustable by means of variable capacitor C48, and has a tuning range of approximately 0° to 160° of phase shift. For a given input signal amplitude, the output signal amplitude is constant, regardless of the phase shift introduced. The output of this circuit is the signal injected into the LM1889 circuits.

CONNECTING TO THE HCS II

Now that you have a working terminal circuit for overlaying text onto live video, you still need to

connect it to the HCS II network. In order to do this, you will need a serial interface compatible with the network, some software that handles network message formats, and software that interprets network messages and creates responses or actions (or both) to those HCS II network messages.

This particular design includes both an RS-232 and an RS-485 interface. U1, a MAX232, provides the RS-232-to-TTL conversion for both the transmitter and receiver. U16, a 75176, provides the RS-485-to-TTL conversion. JP1 connects the receiver pin of the 87C054 to either the RS-232 or RS-485 interfaces. The transmitter pin of the 87C054 connects to both the RS-232 and RS-485 interfaces. One pin of the 87C054, P3.5, controls the driver enable of the 75176, allowing for selective talking on the HCS II network. JP3 provides for termination of the network.

"But, wait a minute! The 87C054 doesn't have a UART," you say. True. There is no built-in UART on the 87C054 and the part does not have a transmitter pin or receiver pin.

Add Text Overlay to Any Video Display

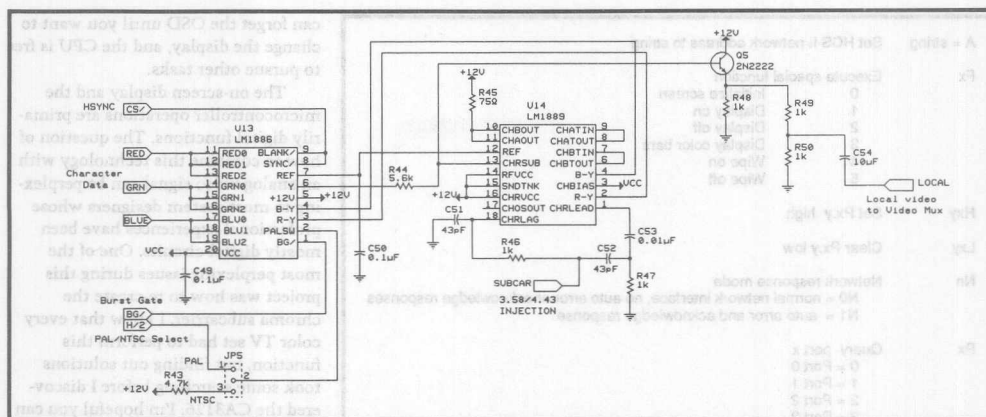


Figure 4e—Encoding the RGB data into baseband video is accomplished with an LM1886, which contains three DACs, and an LM1889, which accepts the regenerated chroma subcarrier, modulates the R-Y and B-Y signals, and produces baseband video.

In this application, the serial data transmission and reception has been performed in software. The routine that handles serial transmission and reception was taken from the Signetics BBS ([800] 451-6644). It was originally designed for the 87C751 and had to be slightly modified to operate with one of the 87C054's timer/counters. The technique is often called "bit banging" and has the advantage of saving some hardware if you can afford the necessary time required of the software.

NETWORK PROTOCOL PROCESSING

As I indicated earlier, in addition to the serial interface software, you need code that handles network message formats. The code starts by waiting until either a "#" or an "!" is received, either of which indicates the start of a network message, then the entire message is stored in a buffer.

Once the carriage return has been stored, the beginning character of the message is checked to see whether the message includes a checksum. If the message does not contain a checksum, the packet is assumed to be valid and the contents of the packet are processed. If a checksum is included, then the VERIFY routine is called to perform a checksum calculation on the packet. If the checksum matches, the packet is processed; otherwise, it is ignored and I return to waiting for the

next network message.

My original plans for handling network checksums included a checksum generator for sending network responses and a checksum checker for received messages. However, when I flowcharted the needs of both routines, I found that an awful lot of the logic was common to both. I went back and looked at the suggestions that Ed Nisley had provided for handling the checksums and understand now why he made those suggestions. My VERIFY routine's logic is based on Ed's previous work.

The VERIFY routine performs two functions. First, it takes the checksum digits in the packet, converts them to binary numbers, and stores them in temporary variables. Next, the checksum digits are replaced with ASCII zeros and the checksum of the string is calculated. If the checksum matches, the error flag, CHKERR, is cleared; otherwise, it is set. The checksum that was calculated is converted to ASCII and stuffed into the checksum digits position, replacing the ASCII zeros.

To prepare a string for transmission, all that is necessary is to stuff the message in the buffer with the checksum digits set to ASCII zeros and call the VERIFY routine. To check a message for correct checksum, simply call the VERIFY routine and check the CHKERR flag on return.

Once the checksum verification (if required) has been performed, you still need to process the packet to see if it belongs to this terminal, and if it does, then you need to determine what action the network controller is asking you to take.

The PROCESS routine first scans the packet, converting characters into upper case until the end of the packet has been reached. Next, the first character is examined to determine if the packet has checksums or not and a pointer is set to the NODEID position of the packet. The NODEID in the packet is compared with the NODEID variable. If there is no match, the packet is ignored and you wait for the next network message. If it does belong to this terminal, you can process the body of the network message.

NETWORK COMMANDS AND SYNTAX

The real essence of a network message is to carry a command from the network controller to the terminal or carry a response from the terminal back to the network controller. Table 1 shows the syntax of the commands available for operating the TV-Link terminal. These commands allow the HCS II Supervisory Controller to manipulate ports on the 87C054, format text for display, implement special built-in display functions such as color bars, and to read and write

Add Text Overlay to Any Video Display

A = string	Set HCS II network address to string
Fx	Execute special function
	0 Initialize screen
	1 Display on
	2 Display off
	3 Display color bars
	4 Wipe on
	5 Wipe off
Hxy	Set Px.y high
Lxy	Clear Px.y low
Nn	Network response mode
	N0 = normal network interface, no auto error or acknowledge responses
	N1 = auto error and acknowledge response
Px	Query port x
	0 = Port 0
	1 = Port 1
	2 = Port 2
	3 = Port 3
Px=nn	Write to port x where nn= two-digit hex value
	0 = Port 0
	1 = Port 1
	2 = Port 2
	3 = Port 3
Rx	Query register x; returns two-digit hex number
	0 = OSDT (contents undefined)
	1 = OSAT
	2 = OSCON
	3 = OSORG
	4 = OSMOD
	5 = Default char. attribute
	6 = Default background space attr
	7 = Default NEWLINE attribute
Rx = nn	Write to register x; for use from outside of a string of text; writes to these registers from within a string; should use the \Wxnn command
	0 = OSDT
	1 = OSAT
	2 = OSCON
	3 = OSORG
	4 = OSMOD
	5 = Default char. attribute
	6 = Default background space attr
	7 = Default NEWLINE attribute
S= string	String for OSD display; can include escape sequences for text formatting, color, selection, etc.
Wxnn	Write to register x; for use within a string; functionally equiv. to the Rx = nn command
Special characters for use within a string of text	
\E	End of Display at current position
\B	Background Space
\S	Split Background Space
\N	NEWLINE

Table 1—The set of supported commands resembles that of most of the other HCS II network modules.

OSD registers directly, giving full control of the OSD to the HCS II.

CONCLUSIONS

Developing this application was interesting and enjoyable. It also

presented some challenges.

The 87C054 proved well suited to this application in large part because of the 80C51 core and that the OSD is independent of the CPU. Once characters have been written to the OSD, you

can forget the OSD until you want to change the display, and the CPU is free to pursue other tasks.

The on-screen display and the microcontroller operations are primarily digital functions. The question of how to combine this technology with an analog video signal can be perplexing to most system designers whose professional experiences have been mostly digital circuits. One of the most perplexing issues during this project was how to re-create the chroma subcarrier. I knew that every color TV set had to perform this function, but finding out solutions took some searching before I discovered the CA3126. I'm hopeful you can profit from my experiences on this project. ☐

My thanks to Herb Kniess and George Ellis of Signetics for their help. Thanks also to Greg Goodhue from Signetics, who wrote the software-based UART code for the 87C751 that I modified for this project.

Bill Houghton is an Applications Engineer at Signetics specializing in 80C51-based microcontrollers.

SOFTWARE

Software for this article is available from the Circuit Cellar BBS and on Software On Disk for this issue. Please see the end of "ConnecTime" in this issue for downloading and ordering information.

SOURCES

Requests for literature on Signetics/Philips microcontrollers including the "80C51-Based 8-Bit Microcontroller Data Handbook" may be directed to Sharon Baker at (408) 991-3518.

Contact Bill Houghton at (408) 991-3560 with technical questions specific to the 87C054 and for information on the availability of a PC board and components for this project.

©Circuit Cellar INK, The Computer Applications Journal. Reprinted by permission.